Herbert Schildt

C++ manual complet

Traducerea
Mihai Dăbuleanu

Teora

Titlul original: C++ THE COMPLETE REFERENCE
Traducere după ediția originală în limba engleză, publicată
de Osborne McGraw-Hill.

Copyright © 1995 McGraw-Hill Book Company International (UK) Limited
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic or mechanical, photocopying, recording or otherwise, without prior permission of McGraw-Hill Book Company International (UK) Limited.

Copyright © 1997 **Teora**
Prima ediție: 1997
Retipărită: aprilie 1998
Toate drepturile asupra versiunii în limba română aparțin Editurii **Teora**.
Reproducerea integrală sau parțială a textului sau a ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al Editurii **Teora**.

Distribuție

București: B-dul Al. I. Cuza nr. 39; tel./fax: 222.45.33
Sibiu: Șos. Alba Iulia nr. 40; tel.: 069/21.04.72; fax: 069/23.51.27
Bacău: Calea Mărășești nr. 5; tel./fax: 034/18.18.26

Teora – Cartea prin poștă

CP 79-30, cod 72450 București, România
Tel./Fax: 252.14.31

Teora

CP 79-30, cod 72450 București, România
Fax: 210.38.28

NOT 2394 CAL C++, MANUAL COMPLET
ISBN 973-601-595-5
Printed in Romania

CUPRINS

Prefață xvii

Partea I
Baza limbajului C++: Limbajul C

1. **O privire de ansamblu asupra limbajului C** 3
   Originile limbajului C++ 4
   C este un limbaj de nivel mediu 4
   C este un limbaj structurat 6
   C este un limbaj al programatorului 7
   Forma unui program în C 9
   Biblioteca și editarea legăturilor 10
   Compilarea independentă 11
   Utilizarea unui compilator de C++ pentru a compila programe în C 12

2. **Expresii** 13
   Cele cinci tipuri de date de bază 14
   Modificarea tipurilor de bază 15
   Nume de identificatori 16
   Variabile 17
   Unde se declară variabilele 17
   Variabile locale 17
   Parametri formali 20
   Variabile globale 21
   Modelatori de acces 22
   const 22
   volatile 24

După cum ştiţi, pointerii şi rudele lor, matricele, sunt importanţi în limbajul C. De aceea, nu va fi o surpriză că aceştia rămân la fel de importanţi şi pentru facilităţile extinse asigurate de către C++. De fapt, pointerii sunt atât de importanţi în C++ încât a fost adăugată o nouă formă de pointeri, numită *referinţă*. Acest capitol studiază matricele, pointerii şi referinţele, relativ la obiecte.

Matrice de obiecte

Este posibil ca în C++ să aveţi matrice de obiecte. Sintaxa pentru declararea şi utilizarea unei astfel de matrice este exact aceeaşi ca şi pentru oricare alt tip de variabilă. De exemplu, acest program foloseşte o matrice de obiecte formată din trei elemente:

```
#include <iostream.h>

class cl {
   int i;
public:
   void pune_i(int j) {i=j;}
   int da_i() {return i;}
};

main()
{
   cl ob[3];
   int i;

   for(i=0; i<3; i++) ob[i].pune_i(i+1);

   for(i=0; i<3; i++)
         cout << ob[i].da_i() << "\n";

   return 0;
}
```

Programul afişează pe ecran numerele **1**, **2** şi **3**.

Dacă o clasă defineşte un constructor cu parametri, puteţi iniţializa fiecare obiect din matrice indicând o listă de iniţializare în acelaşi mod în care o faceţi pentru alte tipuri de matrice. Însă forma exactă a listei de iniţializare depinde de numărul de parametri ceruţi de funcţia constructor a obiectului. Pentru obiectele al căror constructor are un singur parametru, puteţi pur şi simplu să specificaţi o listă de valori iniţiale folosind sintaxa normală de iniţializare a matricelor. Fiecare valoare din listă este transmisă în ordine funcţiei constructor pe măsură ce este

creat fiecare element al matricei. De exemplu, iată o versiune uşor diferită a programului precedent, ce foloseşte iniţializarea:

```
#include <iostream.h>
class cl {
   int i;
public:
   cl(int j) {i=j;} // constructor
   int da_i() {return i;}
};
main()
{
   cl ob[3] = {1, 2, 3}; // initializare
   int i;
   for(i=0; i<3; i++)
         cout << ob[i].da_i() << "\n";
   return 0;
}
```

Şi acest program afişează pe ecran numerele **1**, **2** şi **3**.

Dacă un constructor de obiecte necesită două sau mai multe argumente, atunci va trebui să utilizaţi o formă de iniţializare puţin diferită, prezentată aici:

```
#include <iostream.h>

class cl {
   int h;
   int i;
public:
   cl(int j, int k) {h=j; i=k; } // constructor
   int ia_i() {return i; }
   int ia_h() {return h; }
};

main()
{
   cl ob[3] = {
         cl(1, 2),
         cl(3, 4),
         cl(5, 6)
   };    // initializare

   int i;
```

```
    for(i=0; i<3; i++) {
        cout << ob[i].ia_h();
        cout << ",";
        cout << ob[i].ia_i() << "\n";
    }
    return 0;
}
```

Constructorul lui **cl** are, în acest exemplu, doi parametri şi, de aceea, necesită două argumente. Aceasta înseamnă că forma „prescurtată" nu poate fi folosită. În locul ei, utilizaţi forma „lungă" prezentată în exemplu. Desigur, puteţi folosi forma lungă şi în cazurile în care constructorul necesită doar un singur argument. Numai că forma scurtă este mai uşor de utilizat.

Matrice iniţializate / matrice neiniţializate

O situaţie de clasă specială se întâlneşte atunci când doriţi să creaţi atât matrice iniţializate cât şi neiniţializate. Să considerăm următoarea **class**:

```
class cl {
    int i;
public:
    cl(int j) {i=j;}
    int da_i() {return i;}
};
```

Funcţia constructor definită aici de **cl** necesită un parametru. Aceasta înseamnă că orice matrice declarată ca fiind de acest tip trebuie iniţializată, deci următoarea declaraţie de matrice este greşită:

```
cl a[9]; // eroare, constructorul necesita initializare
```

Motivul pentru care instrucţiunea nu este corectă (aşa cum este definită **cl**) constă în aceea că această clasă ar avea un constructor fără parametri, deoarece nu este specificată nici o iniţializare. Totuşi, aşa cum s-a afirmat, **cl** nu are constructor de parametri. Deoarece nu există nici un constructor valabil care să corespundă acestei declaraţii, compilatorul va afişa o eroare.

Pentru a rezolva această problemă, este necesar să supraîncărcaţi funcţia constructor, adăugând încă una care nu preia nici un parametru. În acest fel, sunt admise atât matricele care sunt iniţializate, cât şi acelea care nu sunt iniţializate. (Supraîncărcarea este discutată în detaliu în **Capitolul 14**.) Iată, de exemplu, o versiune îmbunătăţită a lui **cl**:


```
class cl {
    int i;
public:
    cl() {i=0;} // apelare pentru matrice neinitializate
    cl(int j) {i=j;} // apelare pentru matrice initializate
    int da_i() {return i;}
};
```

Fiind date aceste **class**, sunt permise ambele instrucţiuni care urmează:

```
cl a1[3] = {3, 5, 6};     // initializat
cl a2[34];      // neinitalizat
```

Pointeri către obiecte

Exact aşa cum puteţi avea pointeri către alte tipuri de variabile, puteţi avea pointeri către obiecte. Când doriţi acces la membrii unei clase cu ajutorul unui pointer către un obiect, folosiţi operatorul săgeată (->) în locul operatorului punct. Următorul program ilustrează cum se poate căpăta acces la un obiect, dacă există un pointer către el:

```
#include <iostream.h>

class cl {
    int i;
public:
    cl(int j) {i=j;}
    int da_i() {return i;}
};

main()
{
    cl ob(88), *p;

    p = &ob; // da adresa lui ob

    cout << p->da_i(); // foloseste -> pentru a apela da_i()

    return 0;
}
```

După cum ştiţi, atunci când este incrementat un pointer, el indică spre următorul element de acelaşi tip cu al său. De exemplu, un pointer de tip întreg va indica spre următorul întreg. În general, întreaga aritmetică a pointerilor este relativă la tipul de bază al acestora, la tipul de date către care indică pointerul. Acelaşi lucru este valabil pentru pointeri spre obiecte. De exemplu, următorul program foloseşte un pointer pentru a obţine acces la toate trei elementele matricei **ob** după ce i-a fost atribuită adresa de început din **ob**.

```
#include <iostream.h

class cl {
    int i;
public:
    cl() {i=0;}
    cl(int j) {i=j;}
    int da_i() {return i;}
};

main()
{
    cl ob [3] = {1, 2, 3};
    cl *p;
    int i;

    p = ob; // preia inceputul matricei
    for(i=0; i<3; i++) {
        cout << p->da_i() << "\n";
        p++; // indica spre urmatorul obiect
    }

    return 0;
}
```

Puteţi să atribuiţi unui pointer adresa unui membru public al unui obiect şi apoi să aveţi acces la acel membru folosind pointerul. De exemplu, acesta este un program în C++ care afişează pe ecran numărul **1**:

```
#include <iostream.h>

class cl {
public:
    int i;
    cl(int j) {i=j;}
};

main()
{
    cl ob(1);
    int *p;

    p = &ob.i; // preia adresa lui ob.i

    cout << *p; // are acces la ob.i prin p

    return 0;
}
```

Deoarece **p** indică spre un întreg, el este declarat ca un pointer de tip întreg. În această situaţie este irelevant că **i** este un membru al obiectului **ob**.

Pointeri de verificare a tipului în C++

Trebuie să înţelegeţi un lucru important despre pointeri în C++: puteţi să atribuiţi un pointer altuia doar dacă cei doi au tipuri compatibile. De exemplu, dacă se dă

```
int *pi;
float *pf;
```

următoarea atribuire este ilegală în C++:

```
pi = pf; // eroare -- nepotrivire de tipuri
```

Desigur, puteţi să eliminaţi orice incompatibilitate folosind un modelator, dar procedând astfel violaţi mecanismul de verificare a tipului din C++.

***NOTĂ:** Verificarea mai strictă a tipurilor în C++, atunci când sunt implicaţi pointeri, diferă de C, în care puteţi atribui orice tip de valoare oricărui pointer.*

Pointerul *this*

Când este apelată o funcţie membru, i se pasează automat un argument implicit, care este un pointer către obiectul care a generat apelarea (obiectul care a invocat funcţia). Acest pointer este numit **this**. Pentru a-l înţelege pe **this**, să luăm un

program care creează o clasă numită **putere**, ce calculează rezultatul dat de ridicarea unui număr la o putere:

```
#include <iostream.h>

class putere {
    double b;
    int e;
    double val;
public:
    putere(double baza, int exp);
    double da_put() {return val;}
};

putere::putere(double baza, int exp)
{
    b = baza;
    e = exp;
    val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) val = val * b;
}

main()
{
    putere x(4.0, 2), y(2.5, 1), z(5.7, 0);

    cout << x.da_put() << " ";
    cout << y.da_put() << " ";
    cout << z.da_put() << "\n";

    return 0;
}
```

La membrii unei **class** se poate căpăta acces direct din cadrul unei funcţii membru, fără nici un specificator de obiect sau de clasă. De aceea, instrucţiunea

```
b = baza;
```

din interiorul funcţiei **putere()**, comandă ca valoarea conţinută în **baza** să fie atribuită unei copii a lui **b** asociată obiectului care a generat apelarea. Totuşi, aceeaşi instrucţiune poate fi scrisă şi astfel:

```
this->b = baza;
```


Amintiţi-vă că pointerul **this** indică spre obiectul care a apelat **putere()**. Astfel, **this->b** se referă la copia b pentru acel obiect. De exemplu, dacă **putere()** este apelată de **x** (ca în **x(4.0, 2)**), atunci **this** din instrucţiunea precedentă indică spre **x**. Reţineţi că scrierea instrucţiunii fără **this** este doar o prescurtare.

Iată întreaga funcţie **putere()** scrisă folosind pointerul **this**:

```
putere::putere(double baza, int exp)
{
    this->b = baza;
    this->e = exp;
    this->val = 1;
    if(exp==0) = 1;
    for( ; exp>0; exp--)
        this->val = this->val * this->b;
```

Nici un programator de C++ nu va scrie, de fapt, **putere()** aşa cum tocmai am arătat deoarece nu se câştigă numic, iar forma prescurtată este mai uşoară. Totuşi, pointerul **this** este foarte important la supraîncărcarea operatorilor (vedeţi Capitolul 14) şi ori de câte ori o funcţie membru trebuie să utilizeze un pointer către obiectul care a apelat-o.

Amintiţi-vă că pointerul **this** este transmis automat către toate funcţiile membru. De aceea, **da_put()** poate fi rescrisă astfel:

```
double da_put() {return this->val;}
```

În acest caz, dacă **da_put()** este apelată astfel:

```
y.da_put();
```

atunci **this** va indica spre obiectul y.

Încă două lucruri despre **this**. Primul, funcţiile **friend** nu sunt membri ai clasei şi, de aceea, nu le sunt pasaţi pointeri **this**. Al doilea, funcţiile membre **static** nu au un pointer **this**.

Pointeri către tipuri derivate

În general, un pointer de un anumit tip nu poate indica spre un obiect de un tip diferit. Totuşi, există o excepţie importantă la această regulă care se referă doar la clasele derivate. Pentru început, să presupunem că avem două clase numite **B** şi **D**. Să mai presupunem că **D** este derivat din clasa de bază **B**. În această situaţie, un pointer de tip **B*** poate să indice şi spre un obiect de tip **D**. Mai general, un pointer din clasa de bază poate să fie folosit ca un pointer spre un obiect din oricare clasă derivată din acea bază.

Chiar dacă un pointer din clasa de bază poate indica spre un obiect derivat, reciproca nu este adevărată. Un pointer de tip D* nu poate indica spre un obiect de tip B. Mai mult, chiar dacă puteţi să folosiţi un pointer din bază pentru a indica un obiect derivat, puteţi avea acces doar la membrii de tipul derivat care au fost importaţi din bază. Aceasta înseamnă că nu veţi fi putea avea acces la nici unul din membrii adăugaţi de clasa derivată. (Puteţi însă să convertiţi un pointer din bază într-unul derivat şi să câştigaţi acces deplin la întreaga clasă derivată.)

Iată un scurt program care ilustrează această facilitate din C++:

```
#include <iostream.h>

class baza {
    int i;
public:
    void pune_i(int num) {i=num;}
    int da_i() {return i;}
};

class derivat: public baza {
    int j;
public:
    void pune_j(int num) {j=num;}
    int da_j() {return j;}
};

main()
{
    baza *bp;
    derivat d;

    bp = &d; // pointerul din baza indica spre obiectul derivat

    // acces la obiectul derivat folosind pointerul din baza
    bp->pune_i(10);
    cout << bp->da_i() << " ";

/* Urmatoarele comenzi nu vor functiona. Nu puteti avea
   acces la un element al clasei derivate folosind un
   pointer din clasa de baza.
    bp->pune_j(88); // eroare
    cout << bp->da_j(); // eroare

*/
```



```
  return 0;
}
```

După cum puteţi vedea, un pointer din bază este folosit pentru a oferi acces la un obiect dintr-o clasă derivată.

Pentru a avea acces la un membru al clasei derivate prin intermediul unui pointer de bază, puteţi să-l convertiţi într-altul, al tipului derivat, chiar dacă acest lucru este considerat ca fiind neprofesionist de către majoritatea programatorilor în C++. De exemplu, acesta este un cod valid în C++:

```
// accesul este acum permis datorita conversiei
((derivat *)bp)->pune_j(88);
cout << ((derivat *)bp)->da_j();
```

Este important să vă reamintiţi că aritmetica pointerilor este relativă la tipul de bază al pointerului. Din acest motiv, pentru un pointer din bază, ce indică spre un obiect derivat, incrementarea nu îl determină să indice spre următorul obiect de tipul derivat. În loc de aceasta, el va indica spre ceea ce crede că este următorul obiect de tipul de bază. De exemplu, următorul program, deşi este corect sintactic, conţine această eroare:

```
#include <iostream.h>

class baza {
    int i;
public:
    void pune_i(int num) {i=num;}
    int da_i() {return i;}
};

class derivat: public baza {
    int j;
public:
    void pune_j(int num) {j=num;}
    int da_j() {return j;}
};

main()
{
    baza *bp;
    derivat d[2];
```

```
    bp = d;

    d[0].pune_i(1);
    d[1].pune_i(2);

    cout << bp->da_i() << " ";
    bp++; // relativ la baza, nu la clasa derivata
    cout << bp->da_i(); // afiseaza o valoare gresita

    return 0;
}
```

Utilizarea pointerilor din bază către tipuri derivate este foarte folositoare când creați polimorfism în timpul rulării prin mecanismul funcțiilor virtuale (vedeți Capitolul 16).

Pointeri către membrii clasei

C++ permite să generați un tip special de pointeri care „indică" generic către un membru al unei clase, nu către un anumit exemplar al acelui membru dintr-un obiect. Acest tip de pointer este numit un pointer către un membru al clasei sau, pe scurt, un *pointer-la-membru.* În C++ un pointer la membru nu este același lucru cu un pointer normal. Un pointer la un membru asigură doar un offset (o poziție) într-un obiect din clasa membrului, unde poate fi găsit acel membru. Deoarece pointerii la membri nu sunt pointeri adevărați, nu li se pot aplica operatorii **.** și **->**. Pentru a avea acces la membrul unei clase prin intermediul unui pointer spre el, va trebui să folosiți operatorii speciali ai pointerilor la membri, **.*** și **->***. Misiunea lor este să vă permită accesul la un membru al unei clase prin intermediul unui pointer către acesta.

Iată un exemplu:

```
#include <iostream.h>

class cl{
public:
    cl(int i) {val=i;}
    int val;
    int val_dubla() {return val+val;}
};

main()
{
    int cl::*date;    // pointer la o data membru
```



```
    int (cl::*func)(); // pointer la o functie membru
    cl ob1(1), ob2(2); // creeaza obiecte

    date = &cl::val; // da offsetul pentru val
    func = &cl::val_dubla; // da offsetul pentru val_dubla()

    cout << "Iata valorile: ";
    cout << ob1.*date << " " << ob2.*date << "\n";
    cout << "Aici ele sunt dublate: ";
    cout << (ob1.*func)() << " ";
    cout << (ob2.*func)() << "\n";

    return 0;
}
```

În **main()** acest program creează doi pointeri la membri: **date** și **func**. Notați cu atenție sintaxa fiecărei declarații. Când declarați pointeri la membri, trebuie să specificați clasa și să folosiți operatorul de specificare a domeniului. Programul creează, de asemenea, obiectele de tip **cl** numite **ob1** și **ob2**. După cum ilustrează programul, pointerii la membri pot să indice atât funcții cât și date. Apoi programul obține adresele lui **val** și **val_dubla()**. După cum am spus mai devreme, aceste „adrese" sunt, de fapt, offseturi într-un obiect de tip **cl**, la care pot fi găsite **val** și **val_dubla()**. În continuare, pentru a afișa valorile oricăror obiecte de tip **val**, fiecare valoare este accesată prin **date**. În sfârșit, programul folosește **func** pentru a apela funcția **val_dubla()**. Parantezele în plus sunt necesare pentru corecta asociere a operatorului **.***.

Când cereți acces la un membru al unui obiect folosind un obiect sau o referință (discutată mai târziu în acest capitol), trebuie să folosiți operatorul **.***. Dar, dacă utilizați un pointer spre un obiect, va trebui să folosiți operatorul **->***, așa cum se ilustrează în următoarea versiune a programului precedent.

```
#include <iostream.h>

class cl{
public:
    cl(int i) {val=i;}
    int val;
    int val_dubla() {return val+val;}
};

main()
{
    int cl::*date;    // pointer la o data membru
```

```
    int (cl::*func)(); // pointer la o functie membru
    cl ob1(1), ob2(2); // creeaza obiecte
    cl *p1, *p2;

    p1 = &ob1;
    p2 = &ob2;

    date = &cl::val; // da offsetul pentru val
    func = &cl::val_dubla; // da offsetul pentru val_dubla()

    cout << "Iata valorile: ";
    cout << p1->*date << "" << p2->*date << "\n";

    cout << "Aici ele sunt dublate: ";
    cout << (p1->*func)() << " ";
    cout << (p2->*func)() << "\n";

    return 0;
}
```

În această versiune, **p1** şi **p2** sunt pointeri la obiecte de tip **cl**. De aceea, operatorul **->*** este folosit pentru a oferi acces la **val** şi la **val_dubla()**.

Amintiţi-vă că pointerii la membri sunt diferiţi de pointerii spre exemplare efective de elemente ale unui obiect. De exemplu, să considerăm următorul fragment. (Să presupunem că s-a declarat **cl** după cum s-a arătat în programul precedent.)

```
int cl::*d;
int *p;
cl o;

p = &o.val // aceasta este adresa unui val efectiv
d = &cl::val // acesta este offsetul lui val generic
```

Aici, **p** este un pointer la un întreg din interiorul unui obiect *efectiv*, pe când **d** este un simplu offset care indică unde se găseşte **val** în orice obiect de tip **cl**.

Operatorii pointer-la-membru se aplică în situaţii speciale. Ei nu sunt folosiţi curent în programarea de zi cu zi.

Referinţe

C++ conţine o caracteristică ce este legată de pointeri. Aceasta este numită referinţă. O *referinţă* este, în esenţă, un pointer implicit, care acţionează ca un alt nume al unui obiect.

Parametri de referinţă

O utilizare importantă a unei referinţe este să vă permită să creaţi funcţii care folosesc automat transmiterea parametrului prin referinţă şi nu metoda implicită în C++, cea de apelare prin valoare.

După cum ştiţi, pentru a crea o apelare prin referinţă în C, trebuie să pasaţi explicit funcţiei adresa argumentului. De exemplu, să considerăm următorul scurt program, care foloseşte această abordare într-o funcţie numită **neg()**, ce inversează semnul variabilei de tip întreg spre care indică argumentul ei.

```
#include <iostream.h>

void neg(int *i);

main()
{
    int x;

    x = 10;
    cout << x << " este negat ";

    neg(&x);
    cout << x << "\n";

    return 0;
}
void neg(int *i)
{
    *i = -*i;
}
```

În acest program, **neg()** preia ca parametru un pointer către un întreg al cărui semn îl va schimba. De aceea, **neg()** este apelat explicit cu adresa lui **x**. Mai departe, în interiorul lui **neg()**, pentru a avea acces la variabila spre care indică **i**, trebuie folosit operatorul *****. După cum ştiţi, acesta este modul de generare a unei apelări-prin-referire manuale. Dar, în C++, puteţi automatiza această operaţie folosind un parametru de referinţă.

Pentru a crea un parametru de referinţă, precedaţi numele parametrului cu un **&**. Iată cum este declarat **neg()** prin referinţă:

```
void neg(int &i);
```

Asta spune compilatorului să îl transforme pe **i** ca parametru de referinţă. O dată făcută aceasta, **i** devine, practic, un alt nume pentru orice argument folosit la

apelul funcţiei **neg()**. Cu alte cuvinte, i este un pointer implicit care se referă automat la argumentul folosit pentru invocarea funcţiei **neg()**. O dată transformat i ca referinţă, nu mai este necesar (şi nici chiar permis) să aplicaţi operatorul *. De fiecare dată când este folosit, i este implicit o referinţă la argument. Mai mult, la apelul funcţiei **neg()**, nu mai este necesar (şi nici chiar permis) să precedaţi numele argumentului cu operatorul **&**. Compilatorul va face aceasta automat. Iată o versiune cu referinţă a precedentului program:

```
#include <iostream.h>

void neg(int &i); // i este acum o referinta

main()
{
    int x;

    x = 10;
    cout << x << "este negat"

    neg(x); // nu mai este nevoie de operatorul &
    cout << x <<"\n";

    return 0;
}

void neg(int &i)
{
    i = -i; // i este acum o referinta, nu are nevoie de *
}
```

Să recapitulăm: când creaţi un parametru de referinţă, acesta se referă automat la (indică implicit spre) argumentul folosit pentru a apela funcţia. De aceea, instrucţiunea

```
i = -i;
```

operează efectiv asupra lui **x**, nu asupra copiei lui **x**. Nu este necesar să aplicaţi unui argument operatorul **&**. De asemenea, în interiorul funcţiei, parametrul este folosit direct, fără să aibă nevoie de operatorul *.

Este important să înţelegeţi că atunci când atribuiţi o valoare referinţei, de fapt atribuiţi efectiv acea valoare variabilei căreia i se adresează referinţa. În cazul parametrilor funcţiei, aceasta va fi variabila folosită în apelarea funcţiei.

Nu este posibil să modificaţi în interiorul funcţiei ceea ce „indică" parametrul. Aceasta înseamnă că o instrucţiune ca:

```
i++;
```

aflată în interiorul funcţiei **neg()**, incrementează valoarea variabilei folosită pentru apelare. Ea nu determină ca **i** să indice spre o nouă locaţie.

Iată un alt exemplu. Acest program foloseşte parametri de referinţă pentru a inversa între ele valorile variabilelor cu care este apelată funcţia. (Funcţia **schimb()** este exemplul clasic de transmitere a parametrilor prin referinţă.)

```
#include <iostream.h>

void schimb(int &i, int &j);

main()
{
    int a, b, c, d;
    a = 1;
    b = 2;
    c = 3;
    d = 4;

    cout << "a si b: " << a << " " << b << "\n";
    schimb(a, b); // nu este necesar operatorul &
    cout << " a si b: " << a << " " << b << "\n";

    cout << "c si d: " << c << " " << d << "\n";
    schimb(c, d);
    cout << "c si d: " << c << " " << d << "\n";

    return 0;
}
void schimb(int &i, int &j)
{
    int t;

    t = i; // nu este necesar operatorul *
    i = j;
    j = t;
}
```

Acest program afişează următoarele:

```
a si b: 1 2
a si b: 2 1
c si d: 3 4
c si d: 4 3
```

Transmiterea referinţelor către obiecte

În **Capitolul 12** a fost explicat că atunci când un obiect este transmis unei funcţii ca argument, se face o copie a acelui obiect. Mai mult, când se face copia, *nu* este apelată funcţia constructor obişnuită. (În locul acesteia se face o copie exactă a argumentului apelant.) În schimb, când funcţia se încheie, este apelat destructorul copiei. Dacă, din anumite motive, nu doriţi să fie apelată funcţia destructor, transmiteţi obiectul prin referinţă. (Mai târziu, în acest capitol, veţi vedea asemenea exemple.) Când apelaţi prin referinţă, nu se face nici o copie a obiectului. Aceasta înseamnă că nici un obiect folosit ca parametru nu este distrus atunci când se termină funcţia, iar destructorul parametrului nu este apelat. Încercaţi, de exemplu, acest program:

```
#include <iostream.h>

class cl {
    int ex;
public:
    int i;
    cl(int i);
    ~cl();
    void neg(cl &o) {o.i = -o.i;} // nu este creat temporar
};

cl::cl(int num)
{
    cout << "Construieste " << num << "\n";
    ex = num;
}

cl::~cl()
{
    cout << "Distruge " << ex << "\n";
}

main()
{
```



```
    cl o(1);

    o.i = 10;
    o.neg(o);

    cout << o.i << "\n";

    return 0;
}
```

Iată ieşirea programului:

```
Construieste 1
-10
Distruge 1
```

După cum puteţi vedea, funcţia destructor a lui **cl** este apelată o singură dată. Dacă **o** ar fi apelat prin valoare, atunci în interiorul lui **neg()** s-ar fi creat al doilea obiect, iar destructorul ar fi fost apelat a doua oară când obiectul ar fi fost distrus la terminarea lui **neg()**.

Când parametrii sunt transmişi prin referinţă, amintiţi-vă că schimbările obiectului din interiorul funcţiei afectează obiectul apelant.

Returnarea referinţelor

O funcţie poate să returneze o referinţă. Aceasta are efectul neaşteptat de a permite unei funcţii să fie folosită în membrul stâng al unei instrucţiuni de atribuire! De exemplu, să luăm acest program simplu:

```
#include <iostream.h>

char &inloc(int i); // returneaza o referinta

char s[80] = "Va salut";

main()
{
    inloc(2) = 'X'; // atribuie X spatiului de dupa Va
    cout << s;
    return 0;
}

char &inloc(int i)
```

```
    {
        return s[i];
    }
```

Acest program înlocuieşte spaţiul dintre **Va** şi **salut** cu un **X**. Aceasta înseamnă că programul afişează **VaXsalut**. Să vedem cum se realizează aceasta.

După cum am arătat, **inloc()** este declarată ca returnând o referinţă către o matrice de tip caracter. După cum este scris **inloc()**, ea returnează o referinţă către elementul şirului **s**, specificat de argumentul său **i**. Referinţa returnată de **inloc()** este folosită apoi în **main()** pentru a atribui acelui element caracterul **X**.

Referinţe independente

De departe cea mai obişnuită utilizare pentru referinţe este pasarea argumentelor prin referinţă şi obţinerea valorii returnate de funcţie. Totuşi, puteţi declara o referinţă care să fie doar o simplă variabilă. Acest tip de referinţă este numită *referinţă independentă*.

Când creaţi o referinţă independentă, tot ceea ce creaţi este un al doilea nume pentru altă variabilă. Toate variabilele de tip referinţă independentă trebuie iniţializate atunci când le creaţi. Motivul este uşor de înţeles. În afara iniţializării nu puteţi modifica obiectul spre care indică variabila de referinţă. De aceea, ea trebuie iniţializată atunci când este declarată. (În C++, iniţializarea este o operaţie complet separată de atribuire.)

Programul următor ilustrează o referinţă independentă.

```
#include <iostream.h>

main()
{
    int a;
    int &ref = a; // referinta independenta

    a = 10;
    cout << a << " " << ref << "\n";

    ref = 100;
    cout << a << " " << "\n";

    int b = 19;
    ref = b; // aceasta pune valoarea b in a
    cout << a << " " << ref << "\n";

    ref--; // acesta decrementeaza a
```



```
          // nu afecteaza la ce se refera ref
    cout << a << " " << ref << "\n";

    return 0;
}
```

Programul afişează această ieşire:

```
10 10
100 100
19 19
18 18
```

Pentru a vă referi la o constantă, puteţi folosi o referinţă independentă. De exemplu:

```
int &count = 9;
```

determină **count** să indice localizarea valorii 9 în tabelul de constante din programul dvs.

De fapt, referinţele independente sunt de mică valoare practică deoarece fiecare din ele este, ad literam, doar un alt nume pentru altă variabilă. Având două nume care descriu acelaşi obiect, probabil că programul dvs. va fi confuz şi dezorganizat.

Restricţii pentru referinţe

Există mai multe restricţii care se aplică referinţelor. Nu vă puteţi referi la o altă referinţă. Altfel spus, nu puteţi obţine adresa unei referinţe. Nu puteţi să creaţi matrice de referinţe. Nu puteţi să creaţi un pointer spre o referinţă. Nu puteţi să vă referiţi la un câmp de biţi.

O variabilă de tip referinţă trebuie să fie iniţializată când este declarată, doar dacă nu este un membru al unei clase, un parametru de funcţie sau o valoare returnată. Referinţele nule sunt interzise.

O problemă de stil

Când declară variabile tip pointer sau referinţă, unii programatori de C++ folosesc un stil propriu care asociază * sau & cu numele tipului şi nu cu al variabilei. Iată, de exemplu, două declaraţii echivalente funcţional:

```
int& p; // & asociat cu tipul
int &p; // & asociat cu variabila
```

Asocierea lui * sau & cu numele tipului reflectă dorinţa unor programatori ca C++ să conţină separat un pointer pentru tip. Dar necazul asocierii operatorilor & şi * cu numele tipului, şi nu cu cel al variabilei, este că, în conformitate cu sintaxa formală din C++, nici & şi nici * nu sunt distributivi într-o listă de parametri. De aceea, se pot crea uşor declaraţii greşite. Următoarea declaraţie creează, de exemplu, *unul, nu doi* pointeri de tip întreg. Aici, **b** este declarat ca un întreg (nu ca un pointer de tip întreg) deoarece, conform sintaxei din C++, când este folosit într-o declaraţie, * (sau &) este ataşat variabilei individuale pe care o precede, nu tipului căruia îi urmează.

```
int* a, b;
```

Problema acestei declaraţii este că, văzând mesajul, ni se sugerează că atât **a** cât şi **b** sunt de tip pointer, chiar dacă, de fapt, doar **a** este un pointer. Această confuzie vizuală nu păcăleşte doar programatorii novici în C++, ci uneori şi pe cei cu experienţă.

Este important să înţelegeţi că, în ceea ce priveşte compilatorul de C++, nu are importanţă dacă scrieţi **int *p** sau **int* p**. De aceea, dacă doriţi să asociaţi * sau & cu tipul şi nu cu variabila, sunteţi liberi să o faceţi. Totuşi, pentru a evita confuzia, această carte va continua să asocieze * şi & cu variabilele pe care le modifică şi nu cu tipurile lor.

Operatorii de alocare dinamică din C++

În C, alocarea dinamică a memoriei este realizată prin funcţiile **malloc()** şi **free()**. De dragul compatibilităţii, funcţiile de alocare dinamică din C sunt valabile şi în C++. Dar, C++ asigură un sistem propriu alternativ de alocare dinamică, bazat pe doi operatori: **new** şi **delete**. După cum veţi vedea, există avantaje substanţiale ale facilităţilor de C++ pentru alocarea dinamică a memoriei.

Operatorul **new** returnează un pointer către memoria alocată. Ca şi **malloc()**, **new** alocă memorie din heap (zona de memorie liberă). Dacă nu există memorie suficientă pentru a acoperi alocarea cerută, se returnează un pointer null. Operatorul **delete** eliberează memoria alocată anterior prin utilizarea operatorului **new**. Formele generale pentru **new** şi **delete** sunt:

p_var = new *tip;*
delete *p_var*;

Aici, *p_var* este o variabilă de tip pointer care primeşte un pointer spre memoria care este suficient de mare pentru a păstra un element de tipul *tip*. Iată, de exemplu, un program care alocă memorie pentru a înregistra un întreg:


```
#include <iostream.h>
#include <stdlib.h>

main()
{
    int *p;

    p = new int; // aloca spatiu pentru un int

    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    *p = 100;

    cout << "La " << p << " ";
    cout << " este valoarea " << *p << "\n";

    delete p;
    return 0;
}
```

Operatorul **delete** trebuie să fie folosit doar cu un pointer valid, alocat deja prin utilizarea lui **new**. Folosirea oricărui alt tip de pointer cu **delete** este o operaţiune cu rezultat nedefinit şi, aproape sigur, va determina probleme serioase, cum ar fi căderea sistemului.

Deşi **new** şi **delete** efectuează funcţii similare cu **malloc()** şi **free()**, ele au mai multe avantaje. Mai întâi, **new** alocă automat memorie suficientă pentru a păstra obiectele de tipul specificat. Nu este necesar să folosiţi operatorul **sizeof**. Deoarece mărimea este calculată automat, se elimină orice posibilitate de eroare în această privinţă. Apoi, **new** returnează automat un pointer de tipul specificat. Nu este necesar să folosiţi explicit un modelator de tip, aşa cum o făceaţi când alocaţi memorie cu **malloc()**. În sfârşit, atât **new** cât şi **delete** pot fi supraîncărcate, permiţându-vă să creaţi un sistem de alocare propriu.

Puteţi să iniţializaţi memoria alocată cu o valoare cunoscută, scriind în instrucţiunea **new** o valoare după numele tipului. Iată forma generală a operatorului **new** când include şi iniţializare:

p_var = new *tip_var* (*iniţializator*)

De exemplu, în următorul program întregul căruia i se alocă memorie capătă iniţial valoarea 87.

```
#include <iostream.h>
#include <stdlib.h>

main()
{
   int *p;

   p = new int (87); // initializeaza cu 87

   if(! p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }

   cout << "La " << p << " ";
   cout << "este valoarea " << *p << "\n";

   delete p;

   return 0;
}
```

Folosind **new** cu următoarea formă generală puteţi aloca memorie matricelor:

p_var = new *tip_matrice* [*marime*];

Pentru a elibera memoria pentru o matrice, folosiţi această formă pentru **delete**:

delete [ ] *p_var*;

Aici, [ ] spune operatorului **delete** că va fi eliberată o matrice.
De exemplu, următorul program atribuie memorie pentru o matrice cu zece elemente:

```
#include <iostream.h>
#include <stdlib.h>

main()
{
   int *p, i;

   p = new int [10]; /* aloca memorie pentru o matrice de
                        10 elemente */
```



```
   if(!p) {
         cout << "eroare de alocare\n";
         exit(1);
   }

   for(i=0; i<10; i++)
         p[i];

   for(i=0; i<10; i++;
         cout << p[i] << " ";

   delete [ ] p; // elibereaza memoria pentru matrice

   return 0;
}
```

Observaţi instrucţiunea **delete**. Cum s-a menţionat, când este eliberată memoria alocată cu **new** unei matrice, **delete** trebuie să fie pus în gardă că va fi eliberată o matrice, folosindu-se **[ ]**. (Cum veţi vedea în următorul paragraf, lucrul prezintă importanţă, în special când alocaţi memorie pentru matrice de obiecte.)
Alocării matricelor i se aplică o restricţie: ele nu pot primi valori iniţiale. Aceasta înseamnă că atunci când alocaţi memorie matricelor, nu puteţi specifica o iniţializare.

Alocarea de memorie obiectelor

Obiectelor li se poate aloca memorie dinamic, folosindu-se **new**. Când faceţi aceasta, se creează un obiect şi se returnează un pointer către el. Obiectul creat dinamic se comportă ca şi oricare alt obiect. Când este creat, este apelată funcţia sa constructor (dacă are una). Când este eliberat, se execută funcţia destructor.
Iată un program scurt care creează o clasă numită **bilant** şi care asociază numele unei persoane cu bilanţul său. În interiorul funcţiei **main()** este creat dinamic un obiect de tip **bilant()**.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class bilant {
   double bil_crt;
   char nume[80];
public:
```

```
    void pune(double n, char *s) {
          bil_crt = n;
          strcpy(nume, s);
    }
    void ia_bil(double &n, char *s) {
          n = bil_crt;
          strcpy(s, nume);
    }
};

main()
{
    bilant *p;
    char s[80];
    double n;

    p = new bilant;
    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    p->pune(12387.87, "Ralph Wilson");

    p->ia_bil(n, s);

    cout << s << " are bilantul: " << n;
    cout << "\n";

    delete p;

    return 0;
}
```

Deoarece p conține un pointer către un obiect, operatorul săgeată este folosit pentru a oferi acces la membrii obiectului.

După cum am spus, obiectele alocate dinamic pot avea constructori și destructori. Funcțiile constructor pot fi parametrizate. Să examinăm următoarea versiune a programului anterior:

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
```



```
class bilant {
   double bil_crt;
   char nume[80];
public:
   bilant(double n, char *s) {
         bil_crt = n;
         strcpy(nume, s);
   }
   ~bilant() {
         cout << "Destructor ";
         cout << nume << "\n";
   }
   void ia_bil(double &n, char *s) {
         n = bil_crt;
         strcpy(s, nume);
   }
};

main()
{
   bilant *p;
   char s[80];
   double n;

   // Aceasta versiune foloseste o initializare
   p = new bilant (12387.87, "Ralph Wilson");
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }

   p->ia_bil(n, s);

   cout << s << " are bilantul: " << n;
   cout << "\n";

   delete p;

   return 0;
}
```

Parametrii funcției constructor ai obiectului sunt specificați după numele tipului, la fel ca și pentru alte tipuri de inițializări.

Puteţi să alocaţi memorie pentru matricele de obiecte, dar există o şmecherie. Deoarece nici o matrice căreia i se aplică **new** nu poate fi iniţializată, trebuie să fiţi siguri că, în cazul în care clasa conţine funcţii constructor, una va fi fără parametri. Dacă nu o faceţi, compilatorul de C++ nu va găsi un constructor corespunzător când va încerca să aloce memorie matricei şi nu va compila programul dvs.

În această versiune a programului precedent se alocă memorie unei matrice de obiecte tip **bilant** şi este apelat constructorul fără parametri.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class bilant {
   double bil_crt;
   char nume[80];
public:
   bilant(double n, char *s) {
         bil_crt = n;
         strcpy(nume, s);
   }
   bilant() {} // constructor fara parametri
   ~bilant() {
         cout << "Destructor ";
         cout << nume << "\n";
   }
   void pune(double &n, char *s) {
         bil_crt = n;
         strcpy(nume, s);
   }
   void ia_bil(double &n, char *s) {
         n = bil_crt;
         strcpy(s, nume);
   }
};

main()
{
   bilant *p;
   char s[80];
   double n;
   int i;

   p = new bilant [3]; // aloca memorie pentru intreaga matrice
```



```
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }

   // retineti folosirea punctului, în loc de operatorul sageata
   p[0].pune(12387.87, "Ralph Wilson");
   p[1]. pune(144.00, "A.C. Conners");
   p[2]. pune(-11.23, "I.S. Falitu");

   for(i=0; i<3; i++) {
         p[i].ia_bil(n, s);

   cout << s << " are bilantul: " << n;
   cout << "\n";
   }

   delete [ ] p;

   return 0;
}
```

Un motiv pentru care aveţi nevoie să folosiţi forma **delete [ ]** când distrugeţi o matrice de obiecte alocate dinamic este acela că funcţia destructor poate fi apelată pentru fiecare obiect al matricei.

O notă finală referitoare la new şi **delete**: chiar dacă nu există o regulă care să stabilească acest lucru, este mai bine să nu combinaţi **new** şi **delete** cu **malloc()** şi **free()** în acelaşi program. Nu există nici o garanţie că ele sunt mutual compatibile.

Capitolul 14

Supraîncărcarea funcţiilor şi a operatorilor

C++

Supraîncărcarea funcţiilor şi a operatorilor (overloading) sunt esenţiale pentru programarea în C++. Nu numai că aceste două caracteristici oferă o bază importantă pentru polimorfismul din timpul compilării, dar, de asemenea, ele adaugă limbajului flexibilitate şi extensibilitate. De exemplu, supraîncărcarea operatorilor << şi >> constituie baza abordării I/O în C++.

Acest capitol începe cu supraîncărcarea funcţiilor şi se încheie cu studiul supraîncărcării operatorilor. Chiar dacă este similară supraîncărcării funcţiilor, supraîncărcarea operatorilor introduce mai multe nuanţe ale procesului. De aceea, înainte de a încerca să supraîncărcaţi un operator, trebuie să înţelegeţi pe deplin supraîncărcarea funcţiilor.

Supraîncărcarea funcţiilor

După cum s-a discutat în **Capitolul 11**, supraîncărcarea funcţiilor este pur şi simplu procesul de folosire a aceluiaşi nume pentru două sau mai multe funcţii. Punctul esenţial este, totuşi, acela că fiecare redefinire a funcţiei trebuie să folosească sau tipuri diferite, sau un număr diferit de parametri. Compilatorul nu ştie ce funcţie să apeleze, într-o anumită situaţie, decât prin intermediul acestor diferenţe. De exemplu, următorul program supraîncarcă **functiamea()** folosind tipuri diferite de parametri.

```
#include <iostream.h>

int functiamea(int i); /* acestea difera prin tipurile de
                          parametri */
double functiamea(double i);

main()
{
   cout << functiamea(10) << " "; // apeleaza functiamea(int i);
   cout << functiamea(5.4); // apeleaza functiamea(double i)

   return 0;
}

double functiamea(double i)
{
   return i;
}

int functiamea(int i)
{
   return i;
}
```


Următorul program redefineşte **functiamea()** folosind numere diferite de parametri.

```
#include <iostream.h>

int functiamea(int i); /* acestea difera prin numarul de
                          parametri */
int functiamea(int i, int j);

main()
{
   cout << functiamea(10) << " "; //apeleaza functiamea(int i);
   cout << functiamea(4, 5); //apeleaza functiamea(int i, int j)

   return 0;
}

double functiamea(int i)
{
   return i;
}

int functiamea(int i, int j)
{
   return i*j;
}
```

După cum s-a menţionat, punctul cheie al supraîncărcării funcţiilor este acela că funcţiile trebuie să difere prin tipul sau numărul de parametri. Nu pot fi supraîncărcate două funcţii care diferă doar prin tipul returnat. De exemplu, aceasta este o încercare ilegală de a supraîncărca **functiamea()**:

```
int functia mea(int i);
float functiamea(int i);
/* Eroare: tipurile diferite de returnare sunt insuficiente
   pentru supraîncarcare. */
```

Două declaraţii de funcţii par uneori că diferă, dar, de fapt, nu este aşa. De exemplu, să luăm următoarele declaraţii:

```
void f(int *p);
void f(int p[]); // eroare, *p este acelasi cu p[]
```

Amintiţi-vă că pentru compilator ***p** este acelaşi lucru cu **p[ ]**. De aceea, chiar dacă cele două prototipuri par să difere ca tip sau ca parametri, de fapt, nu este aşa.

Supraîncărcări de funcţii şi ambiguităţi

Puteţi crea cazuri în care compilatorul nu este capabil să aleagă între două (sau mai multe) funcţii supraîncărcate. Când apare această situaţie, ea este numită *ambiguă*. Instrucţiunile ambigue sunt erori, iar programul care conţine o ambiguitate nu este compilat.

Cauza principală a apariţiei ambiguităţilor este, în cele mai multe cazuri, conversia automată a tipului în C++. După cum ştiţi, C++ încearcă să convertească automat argumentele care sunt folosite pentru a apela o funcţie în tipurile argumentelor aşteptate de funcţie. Să luăm, de exemplu, acest fragment:

```
int functiamea(double d);
.
.
.
cout << functiamea('c'); // nu este o eroare, se aplica conversia
```

După cum indică şi comentariul, nu este o eroare, deoarece C++ converteşte automat caracterul **c** în echivalentul său **double**. În C++ există puţine conversii de tip de acest fel care nu sunt admise. Deşi conversiile automate sunt foarte utile, ele constituie încă principala sursă a ambiguităţilor. De exemplu, să luăm următorul program:

```
#include <iostream.h>

float functiamea(float i);
double functiamea(double i);

main()
{
   cout << functiamea(10.1) << " "; /* neambiguu, apeleaza
                                       functiamea(double)*/
   cout << functiamea(10); // ambiguu
   return 0;
}

float functiamea(float i)
{
   return i;
}
double functiamea(double i)
{
   return -i;
}
```


Aici **functiamea()** este supraîncărcată astfel încât să poată prelua argumente ori de tipul **float** ori de tipul **double**. În linia fără ambiguităţi, este apelată **functiamea(double)** deoarece, doar dacă nu sunt specificate explicit ca fiind **float**, toate constantele în virgulă mobilă din C++ sunt automat de tipul **double**. De aceea, această apelare nu este ambiguă. Dar, când **functiamea()** este apelată folosind numărul întreg 10, se introduce o ambiguitate deoarece compilatorul nu are de unde să ştie dacă trebuie convertit în **float** sau în **double**. Aceasta va determina afişarea unui mesaj de eroare, iar programul nu va fi compilat.

După cum ilustrează exemplul precedent, nu supraîncărcarea funcţiei **functiamea()** relativ la **double** şi la **float** este cea care determină ambiguitatea, ci faptul că este apelată cu un argument de tip nedeterminat. Altfel spus, eroarea nu este determinată de supraîncărcarea funcţiei **functiamea()**, ci de apelarea efectivă.

Iată un alt exemplu de ambiguitate determinată de conversia automată a tipului în C++:

```
#include <iostream.h>

char functiamea(unsigned char ch);

char functiamea(char ch);

main()
{
   cout << functiamea('c'); // aceasta apeleaza functiamea(char)
   cout << functiamea(80) << " "; // ambigua

   return 0;
}

char functiamea(unsigned char ch)
{
   return ch-1;
}

char functiamea(char ch)
{
   return ch+1;
}
```

În C++, **unsigned char** şi **char** *nu* sunt implicit ambigue. Totuşi, când este apelată **functiamea()** folosind numărul întreg 88, compilatorul nu ştie ce funcţie să apeleze. Adică, 88 trebuie convertit în **char** sau în **unsigned char?**

O altă cale prin care puteţi crea ambiguităţi este folosirea argumentelor implicite în funcţiile supraîncărcate. (Argumentele implicite sunt prezentate în **Capitolul 21**.) Pentru a vedea cum, să examinăm următorul program:

```
#include <iostream.h>

int functiamea(int i);

int functiamea(int i, int j=1);

main()
{
   cout << functiamea(4, 5) << " "; // neambigua
   cout << functiamea(10); // ambigua

   return 0;
}

int functiamea(int i)
{
   return i;
}
int functiamea(int i, int j)
{
   return i*j;
}
```

Aici, la prima apelare pentru **functiamea()**, sunt specificate două argumente; de aceea, nu apare nici o ambiguitate, şi este apelată **functiamea(int i, int j)**. Dar, când este apelată a doua oară **functiamea()**, apare ambiguitatea, deoarece compilatorul nu ştie dacă să apeleze versiunea pentru **functiamea()** care are un argument, sau să aplice parametrul implicit versiunii care preia două argumente.

Unele tipuri de supraîncărcare a funcţiilor sunt implicit ambigue chiar dacă, la început, ele nu par astfel. Să considerăm, de exemplu, acest program:

```
// Acest program contine o eroare.
#include <iostream.h>

void f(int x);
void f(int &x); // eroare

main()
{
   int a=10;

   f(a); // eroare, care f()?

   return 0;
}

void f(int x)
{
   cout << "In f(int)\n";
}

void f(int &x)
{
   cout << "In f(int &)\n";
}
```


După cum s-a comentat în programul descris, două funcţii nu pot fi supraîncărcate când singura diferenţă dintre ele este aceea că una preia un parametru referinţă şi alta unul normal, apelat prin valoare. În această situaţie, compilatorul nu are cum să ştie de care versiune este vorba la apelare. Amintiţi-vă că nu este nici o diferenţă între felul în care este specificat un argument când este primit de un parametru de referinţă sau de unul de tip valoare.

Anacronisme pentru supraîncărcare

Când a fost creat C++, pentru a crea o funcţie supraîncărcată a fost necesar cuvântul cheie **overload**. Chiar dacă nu mai este necesar şi este considerat caduc, el este acceptat uzual de compilatoarele de C++ din motive de compatibilitate cu vechile programe în C++. (Totuşi, nu este nici o cerinţă care să garanteze aceasta.) Deoarece puteţi întâlni programe vechi, sau poate că vă aflaţi în situaţia în care este disponibil doar un compilator vechi de C++, este bine să ştiţi cum era folosit **overload**. Iată forma sa generală:

overload: *nume-func*;

Aici, *nume-func* este numele funcţiei pe care o veţi supraîncărca. Acest element trebuie să preceadă declaraţiile de supraîncărcare. De exemplu, următoarea linie spune unui compilator de tip vechi că veţi supraîncărca o funcţie numită **test()**:

```
overload test;
```

Deoarece **overload** este un anacronism, ar trebui să evitaţi utilizarea sa în programele de C++ pe care le creaţi.

Supraîncărcarea funcţiilor constructor

În afară de rolul special de iniţializare, funcţiile constructor nu diferă de alte tipuri de funcţii. Aceasta include şi supraîncărcarea. De fapt, funcţiile constructor supraîncărcate se întâlnesc foarte des. De exemplu, să considerăm următorul program, care creează o **class** numită **data**, care conţine o dată calendaristică. Notaţi supraîncărcarea constructorului în două moduri.

```
#include <iostream.h>
#include <stdio.h>

class data {
    int zi, luna, an;
public:
    data(char *d);
    data(int z, int l, int a);
    void arata_data();
};

// Initializeaza sirul folosit.
data::data(char *d)
{
    sscanf(d, "%d%*c%d%*c%d", &zi, &luna, &an);
}

// Initializeaza intregii folositi.
data::data(int z, int l, int a)
{
    zi = z;
    luna = l;
    an = a;
}
void data::arata_data()
{
    cout << zi << "/" << luna;
    cout << "/" << anul << "\n";
}
main()
{
    data ob1(4, 12, 96), ob2("22/10/97");

    ob1.arata_data();
    ob2.arata_data();

    return 0;
}
```


În acest program puteţi să iniţializaţi un obiect de tipul **data** specificând data prin trei cifre pentru zi, lună şi an, sau printr-un şir care conţine data în forma sa generală:

zz/ ll /aa

Cel mai uzual motiv pentru a supraîncărca un constructor este de a permite unui obiect să fie creat folosind cele mai convenabile şi mai naturale metode pentru fiecare circumstanţă. De exemplu, în următoarea funcţie **main()** utilizatorului i se solicită data, care este introdusă în matricea **s**. Acest şir poate fi apoi folosit direct pentru a-l crea pe **d**. Nu este necesar ca el să fie convertit în nici o altă formă. Dar, dacă **data()** nu ar fi supraîncărcată pentru a accepta forma de şir, ar fi trebuit să o convertiţi manual în trei întregi de fiecare dată când creaţi un obiect.

```
main()
{
    char s[80];

    cout << "Introduceti noua data: ";
    cin >> s;

    data d(s);
    d.arata_data();

    return 0;
}
```

În altă situaţie poate fi mai convenabilă iniţializarea unui obiect de tip **data** folosind trei întregi. De exemplu, dacă data este generată printr-o metodă de calcul, atunci cel mai natural şi mai corespunzător constructor ce trebuie utilizat este crearea unui obiect **data** folosind **data(int, int, int)**. Cheia este aceea că, aici, prin supraîncărcarea constructorului pentru **data**, l-aţi făcut mai flexibil şi mai uşor de utilizat. Această creştere de flexibilitate şi de uşurinţă în utilizare sunt importante în special dacă doriţi să creaţi biblioteci de clase care vor fi utilizate de alţi programatori.

NOTĂ: C++ defineşte un tip special de constructor supraîncărcat, numit un constructor de copie, care vă permite să determinaţi cum sunt copiate obiectele în anumite circumstanţe. Constructorii de copie vor fi discutaţi mai departe în această carte.

Găsirea adresei unei funcţii supraîncărcate

După cum ştiţi, în C puteţi să atribuiţi adresa unei funcţii unui pointer şi apoi să apelaţi acea funcţie folosind pointerul. Aceeaşi facilitate există, de asemenea, şi în C++. Dar, datorită supraîncărcării funcţiilor, acest proces este ceva mai complex. Pentru a înţelege de ce, să luăm următoarea instrucţiune, care atribuie adresa unei funcţii numită **functiamea()** unui pointer numit **p**.

```
p = functiamea;
```

Dacă aceasta face parte din C, atunci există o funcţie, şi numai una, numită **functiamea()**, iar compilatorul nu are dificultăţi în a-i atribui adresa sa pointerului **p**. Însă, dacă această instrucţiune face parte din C++, atunci **functiamea()** poate fi supraîncărcată. Presupunând că este aşa, cum va şti compilatorul adresa cărei funcţii să o atribuie pointerului **p**? Răspunsul este că totul depinde de felul în care este declarat **p**. De exemplu, să considerăm acest program:

```
#include <iostream.h>

int functiamea(int a);
int functiamea(int a, b);

main()
{
   int (*fp)(int a); // pointer catre int xxx(int)

   fp = functiamea; // indica spre functiamea(int)

   cout << fp(5);

   return 0;
}

int functiamea(int a)
{
   return a;
}

int functiamea(int a, int b)
{
   return a*b;
}
```

Aşa cum ilustrează programul, **fp** este declarat ca un pointer către o funcţie care returnează un întreg şi care preia un argument de tip întreg. C++ foloseşte această informaţie pentru a selecta versiunea **functiamea(int a)** pentru **functiamea()**.

Dacă **fp** ar fi declarat astfel,

```
int (*fp)(int a, int b);
```

atunci lui i s-ar fi atribuit adresa versiunii **functiamea(int a, int b)**, funcţiei **functiamea()**.

Să recapitulăm: când atribuiţi adresa unei funcţii supraîncărcate unui pointer către o funcţie, cea care determină despre care funcţie este vorba este declararea pointerului. Mai mult, declararea funcţiei de tip pointer trebuie să corespundă exact uneia şi numai uneia dintre declarările funcţiei redefinite.

Supraîncărcarea operatorilor

Strâns legată de supraîncărcarea funcţiilor este supraîncărcarea operatorilor. În C++ puteţi să supraîncărcaţi aproape toţi operatorii, astfel încât ei să efectueze operaţii speciale relativ la clasele pe care le creaţi. De exemplu, o clasă care întreţine o memorie stivă poate să supraîncarce + pentru a efectua o operaţie de încărcare, iar -- pentru una de extragere. Când un operator este supraîncărcat, nu se pierde nici una din semnificaţiile sale originale. Se extinde, în schimb, tipul obiectelor cărora li se poate aplica.

Operatorii se supraîncarcă prin crearea funcţiilor **operator**. O funcţie **operator** defineşte operaţiile specifice pe care le va efectua operatorul supraîncărcat relativ la clasa în care este destinat să lucreze. Funcţiile **operator** pot fi sau nu membri ai clasei în care vor opera. Funcţiile **operator** care nu sunt de tip membru sunt, totuşi, de obicei, funcţii **friend** ale clasei. Felul în care sunt scrise funcţiile **operator** diferă pentru cele de tip membru de cele pentru **friend**. De aceea, fiecare categorie va fi examinată separat, începând cu funcţiile **operator** membre.

Crearea unei funcţii operator membru

Funcţiile **operator** membru au această formă generală:

tip-ret nume-clasa::operator#(*lista-arg*)

```
{
    // operații
}
```

Deseori funcțiile operator returnează un obiect din clasa asupra căreia operează, dar *tip-ret* poate fi de orice tip valid. # este o rezervare de loc. Când creați o funcție **operator**, înlocuiți # cu operatorul. De exemplu, dacă supraîncărcați operatorul /, folosiți **operator/**. Când supraîncărcați un operator unar, *lista-arg* va fi validă. Când supraîncărcați un operator binar, *lista-arg* va conține un singur parametru. (Motivul acestei situații neuzuale va fi clarificat imediat.)

În continuare, este prezentat un exemplu simplu de supraîncărcare de operator. Acest program creează o **class** numită **loc** și stochează valori de latitudine și longitudine. El supraîncarcă operatorul + relativ la această clasă. Examinați cu atenție programul, acordând o atenție specială definirii lui **operator+()**.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() {}
    loc(int lg, int lt) {
          longitud = lg;
          latitud = lt;
    }

    void arata() {
          cout << longitud << " ";
          cout << latitud << "\n";
    }
    loc operator+(loc op2);
};

loc loc::operator+(loc op2)
{
    loc temp;

    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;

    return temp;
}
```



```
main()
{
    loc ob1(10, 20), ob2(5, 30);

    ob1.arata(); // afiseaza 10 20
    ob2.arata(); // afiseaza 5 30

    ob1 = ob1 + ob2;
    ob1.arata(); // afiseaza 15 50

    return 0;
}
```

După cum puteți vedea, **operator+()** are doar un parametru, chiar dacă el supraîncarcă operatorul binar +. (Poate că v-ați așteptat la doi parametri care să corespundă celor doi operanzi ai operatorului binar.) Motivul pentru care **operator+()** are doar un parametru este acela că operandul din stânga lui + este pasat implicit funcției folosind pointerul **this**. Operandul din dreapta este pasat în parametrul **ob2**. Faptul că operandul din stânga este pasat folosind **this** implică, de asemenea, un lucru important: când există operatori binari supraîncărcați, cel care generează apelarea funcției **operator** este obiectul din stânga.

Cum am mai menționat, este obișnuit pentru o funcție **operator** supraîncărcată să returneze un obiect din clasa asupra căreia operează. Făcând aceasta, permiteți operatorului să fie folosit în expresii C++ mai mari. De exemplu, dacă funcția **operator+()** returnează alt tip, această expresie nu ar mai fi validă:

```
ob1 = ob1 + ob2;
```

Pentru ca suma lui **ob1** și **ob2** să fie atribuită lui **ob1**, rezultatul acestei operații trebuie să fie un obiect de tip **loc**.

Mai mult, dacă **operator+()** returnează un obiect de tip **loc**, este posibilă următoarea instrucțiune:

```
(ob1 + ob2).arata(); // afiseaza rezultatul lui ob1+ob2
```

În această situație, **ob1+ob2** generează un obiect temporar care încetează să mai existe după ce se termină apelarea pentru **arata()**.

Este important să înțelegeți că o funcție **operator** poate returna orice tip și că tipul returnat depinde doar de aplicația dvs. efectivă. Numai că, deseori, o funcție **operator** va returna un obiect de clasa asupra căreia operează.

O ultimă problemă relativ la funcția **operator()**: ea nu modifică nici unul dintre operanzi. Deoarece utilizarea obișnuită a operatorului + nu modifică nici un

operand, este normal ca nici versiunea supraîncărcată să nu o facă. (De exemplu, 5+7 face 12, dar nici 5 şi nici 7 nu se modifică.) Tot aşa, în urma aplicării asupra unor obiecte din clasa **loc**, **operator+()** nu ar trebui să modifice nici un operand. Chiar dacă sunteţi liber să efectuaţi orice operaţie doriţi în interiorul funcţiei **operator**, în general cel mai bine este să vă menţineţi în contextul utilizării normale a operatorului.

Următorul program adaugă clasei **loc** trei operatori supraîncărcaţi: -, = şi ++ unar. Fiţi foarte atenţi la modul de definire a acestor funcţii.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() {} // cerut pentru constructii temporare
    loc(int lg, int lt) {
          longitud = lg;
          latitud = lt;
    }

    void arata() {
          cout << longitud << " ":
          cout << latitud << "\n";
    }

    loc operator+(loc op2);
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};
loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;
    return temp;
}
loc loc::operator-(loc op2)
{
    loc temp;
    // observati ordinea operanzilor
    temp.longitud = longitud - op2.longitud;
    temp.latitud = latitud - op2.latitude;
```



```
    return temp;
}
loc loc::operator=(loc op2)
{
    longitud = op2.longitud;
    latitud = op2.latitud;

    return *this; // returneaza obiectul care a generat apelarea
}

loc::operator++()
{
    longitud++;
    latitud++;

    return *this;
}

main()
{
    loc ob1(10, 20), ob2(5, 30), ob3(90, 90);

    ob1.arata();
    ob2.arata();

    ++ob1;
    ob1.arata(); // afiseaza 11 21

    ob2 = ++ob1;
    ob1.arata(); // afiseaza 12 22
    ob2.arata(); // afiseaza 12 22

    ob1 = ob2 = ob3; // atribuire multipla
    ob1.arata(); // afiseaza 90 90
    ob2.arata(); // afiseaza 90 90

    return 0;
}
```

Să studiem pentru început funcţia **operator-()**. Reţineţi ordinea operanzilor pentru scădere. Păstrând semnificaţia scăderii, operandul din dreapta semnului minus este scăzut din operandul din stânga. Deoarece cel care generează apelarea funcţiei **operator-()** este obiectul din stânga, datele din **ob2** trebuie scăzute din

cele spre care indică **this**. Este important să vă reamintiţi care operand generează apelarea funcţiei.

În C++, dacă = nu este supraîncărcat, pentru fiecare clasă pe care o definiţi se creează automat o operaţie implicită de atribuire. Atribuirea implicită este o simplă copiere membru cu membru. Totuşi, supraîncărcând =, puteţi stabili explicit ce atribuire va face pentru o anumită clasă. În acest exemplu, = supraîncărcat face exact acelaşi lucru ca în varianta iniţială, dar în alte situaţii el poate efectua alte operaţii. Reţineţi că funcţia **operator=()** returnează ***this**, care este obiectul care a generat apelarea. Acest lucru este necesar dacă doriţi să folosiţi instrucţiunile de atribuire multiplă, aşa cum este aceasta:

```
ob1 = ob2 = ob3; // atribuire multipla
```

În sfârşit, priviţi definirea lui **operator++()**. După cum puteţi vedea, ea nu are parametri. Deoarece ++ este operator unar, operandul său este transmis implicit folosind pointerul **this**.

Reţineţi că atât **operator=()** cât şi **operator++()** modifică valoarea unui operand. În cazul atribuirii, operandului din stânga (cel care a generat apelarea funcţiei **operator=()**) îi este atribuită o nouă valoare. În cazul lui ++, operandul este incrementat. Cum am spus mai devreme, chiar dacă sunteţi liber să cereţi ca aceste funcţii să efectueze orice doriţi, aproape întotdeauna este mai înţelept ca să rămână consecvente cu semnificaţiile lor originale.

Crearea operatorilor de incrementare şi de decrementare cu prefix şi cu sufix

În programul precedent a fost supraîncărcată doar forma cu prefix a operatorului de incrementare pentru clasa **loc**. În versiunile vechi de C++ nu era posibil să determinaţi dacă ++ şi -- supraîncărcate precedau sau urmau operandul lor. De exemplu, presupunând un obiect numit **O**, aceste două instrucţiuni erau identice:

```
O++;
++O;
```

Însă, versiunile moderne de C++ asigură o cale de determinare dacă un increment sau un decrement este anterior sau posterior operandului lor. Pentru a realiza aceasta, definiţi două versiuni ale funcţiei **operator++()**. Una este definită aşa cum este arătat în programul anterior. Cealaltă este declarată astfel:

```
loc operator++(int x);
```

Dacă ++ precede operandul său, este apelată funcţia **operator++()**. Dacă ++ urmează operandului, este apelată **operator++(int x)** iar **x** are valoarea 0.

Exemplul precedent poate fi generalizat. Iată formele generale pentru funcţiile operator ++ şi -- cu prefix şi cu sufix:

```
// Increment cu prefix
tip operator++() {
    // corpul operatorului cu prefix
}

// Increment cu sufix
tip operator++(intx) {
    // corpul operatorului cu sufix
}

//Decrement cu prefix
tip operator--() {
    // corpul operatorului cu prefix
}

//Decrement cu sufix
tip operator--(int x) {
    corpul operatorului cu sufix
}
```

Supraîncărcarea operatorilor prescurtaţi

Puteţi să supraîncărcaţi oricare dintre operatorii „prescurtaţi" din C++, cum sunt +=, -= şi ceilalţi asemănători lor. De exemplu, această funcţie supraîncarcă += relativ la **loc**:

```
loc loc::operator+=(loc op2)
{
    longitud = op2.longitud + longitud;
    latitud = op2.latitud + latitud;

    return *this;
}
```

Ţineţi minte, atunci când supraîncărcaţi unul dintre aceşti operatori, că de fapt combinaţi o atribuire cu o operaţie de alt tip.

Restricţii la supraîncărcarea operatorilor

Există unele restricţii care se aplică supraîncărcării operatorilor. Nu puteţi modifica precedenţa unui operator. Nu puteţi să modificaţi numărul operanzilor preluaţi de un operator. (Totuşi, puteţi ignora unul.) Funcţiile operator nu pot avea argumente implicite. În sfârşit, următorii operatori nu pot fi supraîncărcaţi:

.   ::   .*   ?

Amintiţi-vă că, practic, puteţi efectua orice în interiorul unei funcţii **operator**. De exemplu, dacă doriţi să supraîncărcaţi operatorul **+** astfel încât el să scrie de zece ori într-un fişier de pe disc **Imi place C++**, puteţi să o faceţi. Totuşi, dacă vă îndepărtaţi mult de semnificaţia implicită a operatorului, riscaţi să vă destructuraţi programul. De exemplu, dacă cineva care citeşte programul dvs. va întâlni o instrucţiune ca **Ob1+Ob2**, se va aştepta să aibă loc o adunare - nu, să zicem, un acces la disc. De aceea, înainte să disociaţi un operator supraîncărcat de înţelesul său implicit, fiţi sigur că aveţi un motiv serios pentru a o face. Un bun exemplu în care disocierea este o reuşită se regăseşte în modul în care C++ supraîncarcă operatorii de I/O **<<** şi **>>**. Deşi operatorii de I/O nu au nici o legătură cu deplasarea biţilor, ei asigură o „cheie" vizuală asupra semnificaţiei lor atât pentru I/O cât şi pentru deplasarea biţilor. Totuşi, în general, când supraîncărcaţi un operator, cel mai bine este să rămâneţi în contextul semnificaţiei sale implicite.

Cu excepţia operatorului **=**, funcţiile **operator** sunt moştenite de orice clasă derivată. Dar, o **class** derivată este liberă să supraîncarce relativ la ea însăşi orice operator (inclusiv cei supraîncărcaţi de clasa de bază).

Supraîncărcarea operatorilor folosind o funcţie friend

Înainte de a privi câteva exemple de operatori mai exotici, cum ar fi **[ ]**, **new** sau **delete**, este bine să facem o mică digresiune în care se examinează supraîncărcarea folosind funcţiile **friend**.

Puteţi supraîncărca un operator relativ la o clasă folosind o funcţie **friend**. Deoarece un prieten nu este membru al clasei, nu are un pointer de tip **this**. De aceea, unei funcţii supraîncărcate de tip **friend operator** i se transmit explicit operanzii. Aceasta înseamnă că un prieten care supraîncarcă un operator binar are doi parametri, iar unul care supraîncarcă un operator unar are un parametru. Când supraîncărcaţi un operator binar, operandul din stânga este pasat în primul parametru iar cel din dreapta în cel de-al doilea parametru al funcţiei **friend operator**.

În următorul program funcţia **operator+()** este declarată ca fiind **friend**:


```
#include <iostream.h>

class loc {
   int longitud, latitud;
public:
   loc() { } // necesara constructia temporara
   loc(int lg, int lt) {
         longitud = lg;
         latitud = lt;
   }

   void arata() {
         cout << longitud << " ";
         cout << latitud << "\n";
   }
   friend loc operator+(loc op1, loc op2); // acum este prieten
   loc operator-(loc op2);
   loc operator=(loc op2);
   loc operator++();
};

// acum, + este supraincarcat folosind functia prieten
loc operator+(loc op1, loc op2)
{
   loc temp;

   temp.longitud = op1.longitud + op2.longitud;
   temp.latitud = op1.latitud + op2.latitud;

   return temp;
}
loc loc::operator-(loc op2)
{
   loc temp;
   // observati ordinea operanzilor
   temp.longitud = longitud - op2.longitud;
   temp.latitud = latitud - op2.latitud;

   return temp;
}
loc loc::operator = (loc op2)
{
   longitud = op2.longitud;
   latitud = op2.latitud;
```

```
    return *this; // returneaza obiectul care a generat apelarea
loc loc::operator++()
{
    longitud++;
    latitud++;

    return *this;
}

main()
{
    loc ob1(10, 20), ob2(5, 30);

    ob1 = ob2 + ob2;
    ob1.arata();

    return 0;
}
```

Există câteva restricţii care se aplică funcţiilor de tip **friend operator.** Prima, nu puteţi supraîncărca operatorii =, (), [ ] sau -> folosind aceste funcţii. A doua, aşa cum se explică în următorul paragraf, când supraîncărcaţi operatorii de incrementare sau decrementare utilizând funcţii **friend**, trebuie să folosiţi un parametru de referinţă.

Folosirea unui friend pentru a supraîncărca ++ sau --

Dacă doriţi să folosiţi o funcţie **friend** pentru a supraîncărca operatorul de incrementare sau de decrementare, trebuie să transmiteţi operandul ca parametru de referinţă, deoarece funcţiile **friend** nu au pointeri **this**. Presupunând că rămâneţi fideli semnificaţiei originale a operatorilor ++ şi --, aceste operaţii implică modificarea operandului asupra căruia lucrează. Dar, dacă supraîncărcaţi aceste operaţii folosind un **friend**, atunci operandul este transmis ca parametru prin valoare. Aceasta înseamnă că o funcţie de tip **friend operator** nu are cum să modifice operandul. Deoarece acestei funcţii nu i se transmite pointerul **this** către operand, ci doar o copie a acestuia, nici o modificare adusă parametrului nu afectează operandul care generează apelarea. Totuşi, puteţi corecta acest lucru specificând parametrul funcţiei **friend operator** ca referinţă. Acest lucru face ca orice modificări aduse parametrului în interiorul funcţiei să afecteze operandul care a generat apelarea. De exemplu, următorul program foloseşte funcţiile **friend** pentru a supraîncărca versiunile cu prefix ale operatorilor ++ şi -- relativ la clasa **loc**.


```
#include iostream.h>

class loc {
   int longitud, latitud;
public:
   loc() { }
   loc(int lg, int lt) {
         longitud = lg;
         latitud = lt;
   }

   void arata() {
         cout << longitud << " ";
         cout << latitud << "\n";
   }

   loc operator=(loc op2);
   friend loc operator++(loc &op);
   friend loc operator--(loc &op);
};

loc loc::operator=(loc op2)
{
   longitud = op2.longitud
   latitud = op2.latitud;
   return *this; // returneaza obiectul care a generat apelarea
}
// acum un prieten - foloseste referinta
loc operator++(loc &op)
{
   op.longitud++;
   op.latitude++;

   return op;
}
// il face prieten pe op -- - foloseste referinta
loc operator--(loc &op)
{
   op.longitud--;
   op.latitud--;

   return op;
}
```

```
main()
{
    loc ob1(10, 20), ob2;

    ob1.arata();
    ++ob1;
    ob1.arata(); // afiseaza 11 21

    ob2 = ++ob1;
    ob1.arata(); afiseaza 12 22

    --ob2;
    ob2.arata(); // afiseaza 11 21

    return 0;
}
```

Dacă doriţi să supraîncărcaţi versiunea cu sufix a operatorilor de incrementare şi de decrementare folosind **friend**, specificaţi simplu un al doilea parametru întreg, fictiv. De exemplu, aceste linii arată prototipul versiunii **friend** cu sufix pentru operatorul de incrementare relativ la **loc**:

```
// versiune prieten cu sufix a lui ++
friend loc operator++(loc &op, int x);
```

Funcţiile friend operator adaugă flexibilitate

În multe cazuri, nu există o diferenţă funcţională între supraîncărcarea unui operator folosind o funcţie **friend** sau una de tip membru. Pentru a păstra cel mai înalt grad de încapsulare, cel mai bine este ca, în aceste cazuri, să supraîncărcaţi folosind funcţiile membre. Totuşi, există o situaţie în care utilizarea unui prieten creşte flexibilitatea unui operator supraîncărcat. Să studiem acum acest caz.

După cum ştiţi, când supraîncărcaţi un operator binar folosind o funcţie membru, obiectul din stânga operatorului este cel ce generează apelarea funcţiei **operator** supraîncărcate. Apoi, în pointerul **this** este transmis un pointer către acest obiect. Acum, presupuneţi o clasă numită **CL**, pentru care adunarea unui obiect cu un întreg este redefinită cu ajutorul unei funcţiei membre **operator**. Având un obiect al acelei clase numit **Ob**, următoarea expresie este validă:

```
Ob +100 // valida
```


În acest caz, **Ob** generează apelarea funcţiei supraîncărcate + şi se efectuează adunarea. Dar, ce se întâmplă dacă expresia este scrisă astfel?

```
100 + Ob // invalida
```

În acest caz, cel care apare în stânga este întregul. Deoarece întregul este un tip încorporat, între acesta şi un tip **Ob** nu este definită nici o operaţie. De aceea, compilatorul nu va compila această expresie. După cum vă puteţi imagina, în unele aplicaţii, să trebuiască mereu ca obiectul să fie poziţionat în stânga este o povară care determină eşecuri.

Soluţia problemei precedente este să supraîncărcaţi adunarea folosind o funcţie **friend**, nu o funcţie membru. În acest caz, funcţiei **operator** îi sunt transmise explicit ambele argumente. De aceea, pentru a permite atât *obiect+întreg* cât şi *întreg+obiect*, supraîncărcaţi pur şi simplu funcţia de două ori - o versiune pentru fiecare situaţie. În acest fel, când supraîncărcaţi un operator folosind două funcţii **friend**, obiectul poate să apară atât în partea stângă cât şi în partea dreaptă a operatorului.

Acest program ilustrează cum sunt folosite funcţiile **friend** pentru a defini o operaţie care implică un obiect şi un tip încorporat.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() { }
    loc(int lg, int lt) {
        longitud = lg;
        latitud = lt;
    }

    void arata() {
        cout << longitud << " ";
        cout << latitud << "\n";
    }

    loc operator+(loc op2);
    friend loc operator+(loc op1, int op2);
    friend loc operator+(int op1, loc op2);
};

loc loc::operator+(loc op2)
{
```

```
    loc temp;
    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;
    return temp;
}

// + este supraincarcat pentru loc + int
loc operator+(loc op1, int op2)
{
    loc temp;
    temp.longitud = op1.longitud + op2;
    temp.latitud = op1.latitud + op2;
    return temp;
}

// + este supraincarcat pentru int + loc
loc operator+(int op1, loc op2)
{
    loc temp;
    temp.longitud = op1 + op2.longitud;
    temp.latitud = op1 + op2.latitud;
    return temp;
}

main()
{
    loc ob1(10, 20), ob2(5, 30), ob3(7, 14);

    ob1.arata();
    ob2.arata();
    ob3.arata();

    ob1 = ob2 + 10; // amandoua
    ob3 = 10 + ob2; // sunt valide

    ob1.arata();
    ob3.arata;

    return 0;
}
```


Supraîncărcarea operatorilor new şi delete

Este posibil ca new şi **delete** să fie supraîncărcate. Puteţi face acest lucru dacă doriţi să folosiţi unele metode speciale de alocare de memorie. De exemplu, puteţi avea nevoie de rutine de alocare care, atunci când s-a epuizat memoria disponibilă (heap), să folosească automat un fişier de pe disc ca memorie virtuală. Indiferent de motiv, este un lucru foarte simplu să supraîncărcaţi aceşti operatori.

Aici sunt prezentate scheletele funcţiilor care redefinesc **new** şi **delete**:

```
void *operator new(size_t marime)
{
    // efectueaza alocarea
    return pointer_la_memorie;
}

void operator delete(void *p)
{
    // memoria libera este indicata de p
}
```

Tipul **size_t** este definit ca un tip apt să conţină cea mai mare porţiune de memorie contiguă care poate fi alocată. **size_t** este de tip întreg fără semn. Parametrul **marime** va conţine numărul de octeţi necesar pentru a memora obiectul pentru care se face alocarea. Funcţia **new** supraîncărcată trebuie să returneze un pointer spre memoria alocată, sau zero, dacă apare o eroare de alocare. Cu excepţia acestor obligaţii, funcţia **new** supraîncărcată poate să facă orice altceva îi cereţi.

Funcţia **delete** primeşte un pointer spre regiunea de memorie pe care trebuie să o elibereze şi eliberează sistemului memoria alocată anterior.

Operatorii **new** şi **delete** pot fi suprapuşi global, astfel încât orice utilizare a lor să apeleze versiunile stabilite de dvs.. De asemenea, ele pot fi supraîncărcate relativ la una sau mai multe clase. Să începem cu un exemplu de supraîncărcare pentru **new** şi **delete** relativ la o clasă. Pentru simplitate, nu va fi folosită o manieră nouă de alocare, ci funcţiile supraîncărcate vor invoca pur şi simplu **malloc()** şi **free()**. (În aplicaţiile dvs. puteţi, desigur, să introduceţi altă metodă, oarecare, de alocare.)

Pentru a supraîncărca operatorii **new** şi **delete** relativ la o clasă, declaraţi, pur şi simplu, funcţiile **operator** supraîncărcate ca membre ale clasei. Iată, de exemplu, operatorii **new** şi **delete** supraîncărcaţi pentru clasa **loc**:

```
#include <iostream.h>
#include <stdlib.h>
```

```
class loc {
    int longitud, latitud;
public:
    loc() {}
    loc(int lg, int lt) {
        longitud = lg;
        latitud = lt;
    }

    void arata() {
        cout << longitud << " ";
        cout << latitud << "\n";
    }

    void *operator new(size_t marime);
    void operator delete(void *p);
};

// new supraincarcat relativ la loc
void *loc::operator new(size_t marime)
{
    cout << "In new al meu\n";
    return malloc(marime);
}

// delete supraincarcat relativ la loc
void loc::operator delete(void *p)
{
    cout << "In delete al meu\n";
    free(p);
}

main()
{
    loc *p1, *p2;
    p1 = new loc (10, 20);
    if(!p1) {
        cout << "Eroare de alocare\n";
        exit(1);
    }

    p2 = new loc (-10, -20);
    if(!p2) {
```



```
        cout << "Eroare de alocare\n";
        exit(1);
    }

    p1->arata();
    p2->arata();

    delete p1;
    delete p2;

    return 0;
}
```

Când **new** şi **delete** sunt supraîncărcaţi relativ la o anumită clasă, utilizarea acestor operatori asupra oricărui alt tip de date determină efectuarea operaţiilor **new** şi **delete** iniţiale. Operatorii suprapuşi se aplică doar acelor tipuri pentru care au fost definiţi. Aceasta înseamnă că, dacă adăugaţi următoarea linie în **main()**, **new** va fi efectuat implicit.

```
int *f = new float; // foloseste new implicit
```

Puteţi redefini global **new** şi **delete** prin supraîncărcarea acestor operatori, în afara oricărei declaraţii de clase. Când aceştia sunt suprapuşi global, sunt ignoraţi **new** şi **delete** impliciţi din C++, iar pentru toate alocările cerute sunt folosiţi noii operatori. Desigur, dacă aţi definit vreo versiune de **new** şi **delete** relativ la una sau mai multe clase, atunci când se alocă memorie obiectelor din clasa pentru care au fost definite, vor fi folosite versiunile specifice acelor clase. Cu alte cuvinte, când sunt întâlnite ori **new** ori **delete**, compilatorul verifică mai întâi dacă au fost definite relativ la clasa asupra cărora operează. Dacă este aşa, vor fi folosite acele versiuni specifice. Dacă nu, C++ foloseşte **new** şi **delete** definite global. Dacă acestea au fost supraîncărcate, atunci sunt utilizate aceste versiuni.

Pentru a vedea un exemplu de supraîncărcare globală pentru **new** şi **delete**, studiaţi acest program:

```
#include <iostream.h>
#include <stdlib.h>

class loc {
    int longitud, latitud;
public:
    loc() {}
    loc(int lg, int lt) {
        longitud = lg;
```

```
        latitud = lt;
    }

    void arata() {
        cout << longitud  << " ";
        cout << latitud << "\n";
    }
};

// new global
void *operator new(size_t marime)
{
    return malloc(marime);
}

// delete global
void operator delete(void *p)
{
    free(p);
}

main()
{
    loc *p1, *p2;
    p1 = new loc (10, 20);
    if(!p1) {
        cout << "Eroare de alocare\n";
        exit(1);
    }
p2 = new loc (-10, 20);
if(!p2) {
    cout << "Eroare de alocare\n";
    exit(1);
}

    float *f = new float;
    // foloseste, de asemenea, new supraincarcat
    if(!f) {
        cout << "Eroare de alocare\n";
        exit(1);
    }

    *f = 10.10;
```



```
    cout <, *f << "\n";

    p1->arata();
    p2->arata();

    delete p1;
    delete p2;
    delete f; // foloseste delete supraincarcat

    return 0;
}
```

Rulaţi acest program pentru a vă dovedi că operatorii **new** şi **delete** încorporaţi au fost, într-adevăr, supraîncărcaţi.

Supraîncărcarea operatorilor new şi delete pentru matrice

Dacă doriţi să puteţi aloca memorie matricelor de obiecte folosind sistemul dvs. propriu de alocare, va trebui să supraîncărcaţi **new** şi **delete** a doua oară. Pentru a aloca şi a elibera memorie pentru matrice, trebuie să folosiţi aceste forme:

```
// Aloca memorie unei matrice de obiecte.
void *operator new[](size_t marime)
{
    // Efectueaza alocarea.
    return pointer_la_memorie;
}

// Delete pentru o matrice de obiecte.
void operator delete[](void *p)
{
    /* Memoria libera este indicata de p.
    Destructor apelat automat pentru fiecare element.
    */
}
```

Când se alocă memorie unei matrice, este apelată automat funcţia constructor a fiecărui obiect al acelei matrice. Când este eliberată memoria, se apelează automat funcţia destructor. Nu este nevoie să asiguraţi un cod explicit pentru a realiza aceste acţiuni.

Următorul program alocă şi eliberează memorie pentru un obiect şi pentru o matrice de obiecte de tipul **loc**.

```
#include <iostream.h>
#include <stdlib.h>

class loc {
   int longitud, latitud;
public:
   loc() {longitud = latitud = 0;}
   loc(int lg, int lt) {
         longitud = lg;
         latitud = lt;
   }

   void arata() {
         cout << longitud << " ";
         cout << latitud << "\n";
   }

   void *operator new(size_t marime);
   void operator delete(void *p);
   void *operator new[](size_t marime);
   void operator delete[](void *p);
};

// new supraincarcat relativ la loc
void *loc::operator new(size_t marime)
{
   cout << "In new al meu\n";
   return malloc(marime);
}

// delete supraincarcat relativ la loc
void loc::operator delete (void *p)"
{
   cout << "In delete al meu\n";
   free(p);
}

// new supraincarcat relativ la loc, pentru matrice
void *loc::operator new[](size_t marime)
{
   cout << "Aloca memorie pentru matrice folosind
          new[] propriu\n";
   return malloc(marime);
```



```
}

// delete supraincarcat relativ la loc, pentru matrice
void loc::operator delete[](void *p)
{
   cout << "Elibereaza memorie din matrice folosind
          delete[] propriu\n";
   free(p);
}

main()
{
   loc *p1, *p2;
   int i;

   p1 = new loc (10, 20); // aloca memorie pentru un obiect
   if(!p1) {
         cout << "Eroare de alocare\n";
         exit(1);
   }

   p2 = new loc [10]; // aloca memorie pentru o matrice
   if(!p2) {
         cout << "Eroare de alocare\n";
         exit(1);
   }

   p1->arata();

   for(i=0; i<10; i++)
         p2[i].arata();

   delete p1; // elibereaza un obiect
   delete [] p2; // elibereaza o matrice

   return 0;
}
```

Supraîncărcarea unor operatori speciali

C++ şi predecesorul său C definesc ca operaţii înscrierea matricelor, apelarea funcţiilor şi „indicare spre". Operatorii care efectuează aceste funcţii sunt [ ], () şi

respectiv ->. Aceşti operatori destul de exotici pot fi supraîncărcaţi în C++, înlesnind câteva utilizări foarte interesante.

Există o restricţie importantă care se aplică acestor trei operatori: ei trebuie să fie funcţii membre care nu sunt de tip **static**; nu pot fi **friend**.

Supraîncărcarea pentru [ ]

În C++, atunci când este supraîncărcat, [ ] este considerat a fi un operator binar. De aceea, forma generală a unei funcţii membru de tip **operator[ ]()** este cea prezentată aici:

```
tip nume-clasa::operator[ ](int i)
{
    // ...
}
```

Practic, nu este necesar ca parametrii să fie de tip **int**, dar o funcţie **operator[]()** este folosită tipic pentru a asigura înscrierea indecşilor într-o matrice şi, de aceea, este folosită, în general, o valoare întreagă.

Dându-se un obiect numit **O**, expresia

```
O[3]
```

se transformă în această apelare a funcţiei **operator[ ]()**:

```
operator[](3)
```

Aceasta înseamnă că valoarea expresiei din operatorul de înscriere este transmisă funcţiei **operator[]()** cu parametrul său explicit. Pointerul **this** va indica spre **O**, obiectul care a generat apelarea.

În următorul program, **untip** declară o matrice formată din trei întregi. Funcţia sa constructor iniţializează fiecare element al matricei cu valoarea specificată. Funcţia **operator[]()** supraîncărcată returnează valoarea matricei, având ca indice valoarea parametrului său.

```
#include <iostream.h>

class untip {
   int a[3];
public:
   untip(int i, int j, int k) {
         a[0] = i;
         a[1] = j;
```



```
         a[2] = k;
   }
   int operator[](int i) {return a[i];}
};

main()
{
   untip ob(1, 2, 3);

   cout << ob[1]; // afiseaza 2

   return 0;
}
```

Puteţi proiecta funcţia **operator[]()** astfel încât **[ ]** să fie folosit atât în partea stângă, cât şi în partea dreaptă a unei instrucţiuni de atribuire. Pentru a face aceasta, specificaţi pur şi simplu valoarea returnată de **operator[]()** ca referinţă. Următorul program face această modificare şi îi arată modul de folosire.

```
#include <iostream.h>

class untip {
   int a[3];
public:
   untip(int i, int j, int k) {
         a[0] = i;
         a[1] = j;
         a[2] = k;
   }
   int &operator[](int i) {return a[i];}
};

main()
{
   untip ob(1, 2, 3);

   cout << ob[1]; // afiseaza 2
   cout << " ";

   ob[1] = 25; // [] in stinga lui =

   cout << ob[1]; // acum afiseaza 25
```

```
    return 0;
}
```

Deoarece **operator[ ]()** returnează acum o referință către elementul matricei cu indicele **i**, el poate fi folosit în partea stângă a unei atribuiri, pentru a modifica un element al matricei. (Desigur, poate fi folosit în continuare la fel de bine și în partea dreaptă.)

Un avantaj al supraîncărcării operatorului [ ] este acela că el permite o cale de utilizare sigură a indicilor matricelor în C++. După cum știți, este posibil ca, în C++, să depășiți în sus sau în jos limitele matricei în timpul rulării fără să primiți un mesaj de eroare.

Însă, dacă stabiliți o clasă care conține matricea și permiteți accesul la ea prin operatorul de înscriere supraîncărcat [ ], atunci puteți intercepta indicii din afara limitei. De exemplu, următorul program adaugă, față de precedentul, o verificare a limitei și dovedește că ea funcționează.

```
// Un exemplu de matrice sigura.
#include <iostream.h>
#include <stdlib.h>

class untip {
    int a[3];
public:
    untip(int i, int j, int k) {
          a[0] = i;
          a[1] = j;
          a[2] = k;
    }
    int &operator[](int i) {return a[i];}
};

// Asigura verificarea limitei pentru untip.
int &untip::operator[](int i)
{
    if(i<0 || i>2) {
          cout << "Eroare de limita\n";
          exit(1);
    }
    return a[i];
}

main()
{
```



```
    untip ob(1, 2, 3);
    cout << ob[1]; // afiseaza 2
    cout << " ";
    ob[1] = 25; // [] apare in stanga
    cout << ob[1]; // afiseaza 25

    ob[3] = 44; /* determina eroare in timpul rularii, 3 in
                   afara limitei */

    return 0;
}
```

În acest program, atunci când se execută instrucțiunea

```
ob[3] = 44;
```

eroarea de depășire a limitelor este depistată de **operator[ ]()**, iar programul se încheie înainte de a se produce vreo pagubă. (În practică, pot fi apelate anumite funcții de tratare a erorilor care rezolvă condițiile de ieșire din limite; nu va trebui ca programul să se încheie).

Supraîncărcarea pentru ()

Când supraîncărcați operatorul de apelare a funcției (), nu creați o nouă cale de apelare a unei funcții. Creați, de fapt, o funcție **operator** căreia îi poate fi transmis un număr arbitrar de parametri. Să începem cu un exemplu. Dându-se declararea funcției **operator** supraîncărcate

```
double operator()(int a, float f, char *s);
```

și un obiect **O** de această **class**, atunci instrucțiunea

```
O(10, 23.34, "hi");
```

este transpusă în următoarea apelare a funcției **operator()**:

```
operator()(10, 23.34, "hi");
```

În general, când supraîncărcați operatorul (), definiți parametrii pe care doriți să îi transmiteți acelei funcții. Când folosiți în programele dvs. operatorul (), argumentele pe care le specificați sunt copiate în acei parametri. Ca de obicei, obiectul care generează apelarea (în acest exemplu, **O**) este indicat de pointerul **this**.

Iată un exemplu de supraîncărcare pentru () relativ la clasa **loc**. El atribuie obiectului căruia i se aplică valoarea celor două argumente pentru longitudine şi latitudine.

```
#include <iostream.h>

class loc {
   int longitud, latitud;
public:
   loc() { }
   loc(int lg, int lt) {
         longitud = lg;
         latitud = lt;
   }

   void arata() {
         cout << longitud << " ";
         cout << latitud << "\n";
   }
   loc operator+(loc op2);
   loc operator()(int i, int j);
};

loc loc::operator()(int i, int j)
{
   longitud = i;
   latitud = j;

   return *this;
}

loc loc::operator+(loc op2)
{
   loc temp;
   temp.longitud = op2.longitud + longitud;
   temp.latitud = op2.latitud + latitud;

   return temp;
}

main()
{
   loc ob1(10, 20), ob2(1, 1);
```



```
   ob1.arata();
   ob1(7,8); // poate fi executata singura
   ob1.arata();

   ob1 = ob2 + ob1(10, 10); // poate fi folosita in expresii
   ob1.arata();

   return 0;
}
```

***REŢINEŢI:** Când supraîncărcaţi (), puteţi folosi orice tip de parametri şi returna orice tip de valoare. Aceste tipuri vor fi dictate de cerinţele programelor dvs.*

Supraîncărcarea pentru ->

Când este supraîncărcat, operatorul pointer -> este considerat un operator unar. Iată modul său de utilizare generală:

obiect->element;

obiect este, aici, obiectul care activează apelarea. Funcţia **operator->()** trebuie să returneze un pointer spre un obiect de clasa asupra căreia operează **operator->()**. *element* trebuie să fie un element accesibil din cadrul obiectului returnat de **operator->()**.

Următorul program ilustrează supraîncărcarea lui -> arătând echivalenţa dintre **ob.i** şi **ob->i** când **operator->()** returnează pointerul **this**.

```
#include <iostream.h>

class clasamea {
public:
   int i;
   clasamea *operator->() {return this;}
};

main()
{
   clasamea ob;

   ob->i = 10; // identic cu ob.i
```

```
    cout << ob.i << " " << ob->i;

    return 0;
}
```

Supraîncărcarea operatorului virgulă

Puteți supraîncărca operatorul virgulă. Virgula este un operator binar și, ca pentru toți operatorii suprapuși, puteți să determinați ca o virgulă supraîncărcată să efectueze orice operație doriți. Totuși, dacă doriți ca virgula supraîncărcată să acționeze într-un mod similar cu operația sa normală, ea trebuie să renunțe la valoarea din termenul său stâng și să atribuie valoarea operației termenului din dreapta. Într-o listă separată prin virgulă trebuie să se renunțe la toți termenii, în afară de cel din extrema dreaptă. După cum știți, acesta este felul în care virgula lucrează implicit în C++.

Iată un program care ilustrează efectul supraîncărcării operatorului virgulă în modul său de operare implicit.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() { }
    loc(int lg, int lt) {
          longitud = lg;
          latitud = lt;
    }

    void arata() {
          cout << longitud << " ";
          cout << latitud << "\n";
    }
    loc operator+(loc op2);
    loc operator,(loc op2);
};

loc loc::operator,(loc op2)
{
    loc temp;

    temp.longitud = op2.longitud;
    temp.latitud = op2.latitude;
    cout << op2.longitud << " " << op2.latitude << "\n";

    return temp;
}
loc loc::operator+(loc op2)
{
    loc temp;

    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;

    return temp;
}

main()
{
    loc ob1(10, 20), ob2(5, 30), ob3(1, 1);

    ob1.arata();
    ob2.arata();
    ob3.arata();

    cout << "\n";

    ob1 = (ob1, ob2+ob2, ob3);

    ob1.arata(); // afiseaza 1 1, valoarea lui ob3

    return 0;
}
```

Acest program afișează următoarea ieșire:

```
10 20
5 30
1 1

10 60
1 1
1 1
```

Observaţi că, deşi se renunţă la valorile operanzilor din partea stângă, fiecare expresie este executată de către compilator, deci va apărea orice efect secundar dorit.

Mai reţineţi că operandul din partea dreaptă este transmis prin **this**, iar valoarea sa este eliminată de către funcţia **operator,()**. Funcţia returnează valoarea din dreapta operaţiei. Aceasta determină ca virgula supraîncărcată să se comporte similar cu operaţia sa implicită. Dacă doriţi să supraîncărcaţi virgula pentru a face altceva, va trebui să modificaţi aceste două caracteristici.

Capitolul 15

Moştenirea

Moştenirea este una dintre pietrele de temelie ale OOP deoarece ea permite crearea clasificărilor ierarhice. Utilizând moştenirea, puteţi să construiţi o clasă generală care defineşte trăsăturile comune ale unui set de elemente corelate. Această clasă poate fi apoi moştenită de alte clase, particulare, fiecare adăugând doar acele elemente care îi sunt proprii.

Pentru a ne menţine în terminologia C++ standard, o clasă care este moştenită este numită o *clasă de bază*. Clasa care moşteneşte este numită *clasă derivată*. Mai departe, o clasă derivată poate fi folosită ca una de bază pentru altă clasă derivată. Astfel se realizează moştenirea multiplă.

Suportul pentru moştenire în C++ este bogat şi flexibil. Moştenirea a fost prezentată în **Capitolul 11**. Aici ea este examinată în detaliu.

Controlul accesului la clasa de bază

După cum ştiţi, când o clasă moşteneşte alta, se foloseşte următoarea formă generală:

class *nume-clasa-derivata* : *acces nume-clasa-de-baza* {
    // corpul clasei
};

Când o clasă moşteneşte alta, membrii clasei de bază devin membri ai clasei derivate. Accesul la membrii clasei din interiorul clasei derivate este determinat prin *acces*. Specificatorul de acces al clasei de bază trebuie să fie **public**, **private** sau **protected**. În cazul în care nu este prezent nici un specificator, el este implicit **private** pentru o clasă derivată de tip **class**. Dacă clasa derivată este o **struct**, atunci, în absenţa unui specificator explicit, accesul este implicit **public**. Să examinăm ramificaţiile utilizării accesului **public** sau **private**. (Specificatorul **protected** este studiat în paragraful următor.)

Când specificatorul de acces al unei clase de bază este **public**, toţi membrii publici ai clasei devin membri publici ai clasei derivate, iar toţi membrii **protected** (protejaţi) ai bazei devin membri protejaţi ai clasei derivate. În toate cazurile, elementele **private** (particulare) ale bazei rămân particulare pentru bază şi nu sunt accesibile membrilor clasei derivate. De exemplu, aşa cum se ilustrează în următorul program, obiectele de tip **derivat** pot fi accesibile direct membrilor publici din **baza**.

```
#include <iostream.h>

class baza {
    int i, j;
public:
    void pune(int a, int b) {i=a; j=b;}
    void arata() { cout << i << " " << j << "\n";}
};

class derivat : public baza {
    int k;
public:
    derivat(int x) {k=x;}
    void aratak() { cout << k << "\n";}
};

main()
{
    derivat ob(3);
    ob.pune(1, 2); // acces la membrul bazei
    ob.arata(); // acces la membrul bazei

    ob.aratak(); // foloseste membrul clasei derivate

    return 0;
}
```


Când clasa de bază este moştenită prin utilizarea specificatorului de acces **private**, toţi membrii publici şi protejaţi ai clasei de bază devin membri privaţi ai clasei derivate. De exemplu, următorul program nici măcar nu va fi compilat deoarece atât **pune()**, cât şi **arata()** sunt acum elemente particulare pentru **derivat**.

```
// Acest program nu va fi compilat
#include <iostream.h>

class baza {
    int i, j;
public:
    void pune(int a, int b) {i=a; j=b;}
    void arata() { cout << i << " " << j << "\n";}
};

// Elementele publice din baza sunt particulare in derivat.
class derivat : private baza {
    int k;
public:
    derivat(int x) {k=x;}
    void aratak() {cout << k << "\n";
};
```

```
main()
{
   derivat ob(3);

   ob.pune(1, 2); // eroare, nu poate avea acces la pune()
   ob.arata(); // eroare, nu poate avea acces la arata()

   return 0;
}
```

*REŢINEŢI: Când un specificator de acces al unei clase de bază este **private**, membrii publici sau protejaţi ai bazei devin membri particulari ai clasei derivate. Aceasta înseamnă că ei sunt încă accesibili membrilor clasei derivate, dar şi secţiunilor din programul dvs. care nu sunt membri nici ai clasei de bază, nici ai celei derivate.*

Moştenirea şi membrii protejaţi

Cuvântul cheie **protected** este inclus în C++ pentru a oferi o flexibilitate mai mare mecanismului de moştenire. Când un membru al clasei este declarat ca fiind **protected**, acel membru nu este accesibil altor elemente ale programului care nu sunt membri ai clasei. Cu o singură excepţie importantă, accesul la un membru protejat este acelaşi ca şi accesul la un membru particular - el este accesibil doar altor membri din aceeaşi clasă cu a sa. Singura excepţie este atunci când membrul protejat este moştenit. În acest caz, un membru protejat diferă esenţial de unul particular.

După cum ştiţi din paragraful anterior, un membru particular al unei clase de bază nu este accesibil altei secţiuni ale programului dvs., inclusiv orice clasă derivată. Dar, membrii **protected** se comportă diferit. În cazul în care clasa de bază este moştenită ca **public**, atunci membrii protejaţi ai acestei clase devin membri protejaţi ai clasei derivate şi, de aceea, accesibili acesteia. Cu alte cuvinte, folosind **protected**, puteţi crea membri ai clasei care sunt particulari în clasa lor, dar care pot fi moşteniţi şi sunt accesibili unei clase derivate. Iată un exemplu:

```
#include <iostream.h>

class baza {
protected:
   int i, j; /* particulari pentru baza, dar accesibili
                pentru derivat */
```



```
public:
   void pune(int a, int b) {i=a; j=b;}
   void arata() { cout << i << " " << j << "\n";}
};

class derivat : public baza {
   int k;
public:
   // derivat poate sa aiba acces la i si j din baza
   void punek() {k=i*j;}

   void aratak() {cout << k << "\n";}
};

main()
{
   derivat ob;

   ob.pune(2, 3); // OK, cunoscut lui derivat
   ob.arata(); // OK, cunoscut lui derivat

   ob.punek();
   ob.aratak();

   return 0;
}
```

Aici, deoarece **baza** este moştenită de **derivat** ca **public**, iar **i** şi **j** sunt declarate ca fiind **protected**, funcţia **punek()** din **derivat** poate să aibă acces la ele. Dacă **i** şi **j** ar fi fost declarate ca fiind **private** în **baza**, atunci **derivat** nu ar fi avut acces la ele, iar programul nu ar fi fost compilat.

Când o clasă derivată este folosită ca o clasă de bază pentru altă clasă derivată, atunci orice membru protejat al clasei de bază iniţiale care este moştenită (ca **public**) de către prima clasă derivată poate fi moştenit, de asemenea, ca **protected** şi de cea de-a doua clasă derivată. De exemplu, următorul program este corect, iar **derivat2** poate, într-adevăr, să aibă acces la **i** şi **j**.

```
#include <iostream.h>

class baza {
protected:
   int i, j;
```

```
public:
    void pune(int a, int b) {i=a; j=b;}
    void arata() { cout << i << " " << j << "\n";}
};

// i si j mosteniti ca protejati.
class derivat1 : public baza {
    int k;
public:
    void punek() {k=i*j;} // legal
    void aratak() {cout << k << "\n";}
};

// i si j mosteniti indirect prin derivat1.
class derivat2 : public derivat1 {
    int m;
public:
    void punem() {m = i-j;} // posibil
    void aratam() {cout << m << "\n";}
};

main()
{
    derivat1 ob1;
    derivat2 ob2;

    ob1.pune(2, 3);
    ob1.arata();
    ob1.punek();
    ob1.aratak();

    ob2.pune(3, 4);
    ob2.arata();
    ob2.punek();
    ob2.punem();
    ob2.aratak();
    ob2.aratam();

    return 0;
}
```

Dacă, totuşi, **baza** ar fi moştenită ca fiind **private**, atunci toţi membrii din **baza** ar deveni membri **private** ai lui **derivat1**, ceea ce înseamnă că ei nu ar fi accesibili

lui **derivat2**. (Totuşi, **i** şi **j** ar continua să fie accesibili lui **derivat1**.) Această situaţie este ilustrată de către următorul program, care conţine erori (şi nu va fi compilat). Comentariile descriu fiecare eroare.

```
// Acest program nu va fi compilat.
#include <iostream.h>

class baza {
protected:
    int i, j;
public:
    void pune(int a, int b) {i=a; j=b;}
    void arata() { cout << i << " " << j << "\n";}
};

// Acum, toate elementele din baza sunt particulare in derivat1.
class derivat1 : private baza {
    int k;
public:
    // ilegal deoarece i si j sunt particulari in derivat1
    void punek() {k=i*j;} // OK
    void aratak() {cout << k << "\n";}
};

// Accesul la i, j, pune() si arata() nu este mostenit.
class derivat2 : public derivat1 {
    int m;
public:
    // ilegal deoarece i si j sunt particulari in derivat1
    void punem() {m = i-j;} // eroare
    void aratam() {cout << m << "\n";}
};

main()
{
    derivat1 ob1;
    derivat2 ob2;

    ob1.pune(1, 2); // eroare, nu poate folosi pune()
    ob1.arata(); // eroare, nu poate folosi arata()

    ob2.pune(3, 4);  // eroare, nu poate folosi pune()
    ob2.arata(); // eroare, nu poate folosi arata()
```

```
    return 0;
}
```

*NOTĂ: Chiar dacă **baza** este moştenită ca fiind **private** în **derivat1**, **derivat1** are încă acces la elementele **public** şi **protected** din **baza**. Totuşi, el nu poate transmite mai departe acest privilegiu.*

Moştenirea protected a clasei de bază

Este posibil să moşteniţi o clasă de bază ca **protected**. Când se face acest lucru, toţi membrii publici şi protejaţi ai clasei de bază devin membri protejaţi ai clasei derivate. Iată un exemplu:

```
#include <iostream.h>

class baza {
protected:
   int i, j; /* particulari pentru baza, dar accesibili
                pentru derivat */
public:
   void puneij(int a, int b) {i=a; j=b;}
   void arataij() { cout << i << " " << j << "\n";}
};

// Mosteneste baza ca protected.
class derivat1 : protected baza {
   int k;
public:
   /* derivat poate avea acces la i si j din baza si la
      puneij(). */
   void punek() {puneij(10, 12); k = i*j;}

   // aici poate avea acces la arataij()
   void aratatot() {cout << k << " "; arataij();}
};

main()
{
   derivat ob;

//  ob.puneij(2, 3); // ilegal, puneij() este membru
//                      protejat al lui derivat
```



```
   ob.punek(); // OK, membru public al lui derivat
   ob.aratatot(); // OK, membru public al lui derivat

// ob.arataij(); // ilegal, arataij() este membru
//                  protejat al lui derivat

   return 0;
}
```

După cum puteţi vedea citind comentariile, chiar dacă **puneij()** şi **arata()** sunt membri publici în **baza**, ei devin membri protejaţi în **derivat** atunci când aceasta îi moşteneşte folosind specificatorul de acces **protected**. Deci, aceasta înseamnă că ele nu vor fi accesibile pentru **main()**.

Moştenirea din clase de bază multiple

Este posibil pentru o clasă derivată să moştenească două sau mai multe clase de bază. Iată, de exemplu, **derivat** moşteneşte atât **baza1**, cât şi **baza2**:

```
// Un exemplu de clase de baza multiple.
#include <iostream.h>

class baza1 {
protected:
   int x;
public:
   void aratax() {cout << x << "\n";}
};

class baza2 {
protected:
   int y;
public:
   void aratay() {cout << y << "\n";}
};

// Mostenire din multiple clase de baza.
class derivat: public baza1, public baza2 {
public:
   void pune(int i, int j) {x=i; y=j;}
};

main()
```

```
{
    derivat ob;

    ob.pune(10, 20); // asigurat prin derivat
    ob.aratax(); // din baza1
    ob.aratay(); // din baza2

    return 0;
}
```

După cum ilustrează exemplul, pentru a moşteni mai mult decât o clasă, folosiţi o listă separată prin virgulă şi aveţi grijă să utilizaţi un specificator de acces pentru fiecare bază moştenită.

Constructori, destructori şi moştenire

Există două mari întrebări care se ridică relativ la constructori şi destructori atunci când este vorba de moştenire. Prima, când sunt apelate funcţiile constructor şi destructor din clasele de bază şi cele derivate? A doua, cum pot fi transmişi parametrii către funcţiile constructor din clasele de bază? Acest paragraf examinează cele două caracteristici foarte importante.

Când sunt executate funcţiile constructor şi destructor

Este posibil ca o clasă de bază, una derivată sau ambele să conţină funcţii constructor şi/sau destructor. Este important să înţelegeţi ordinea în care sunt executate aceste funcţii atunci când este creat un obiect al clasei derivate sau când încetează să mai existe. Examinaţi, pentru început, acest scurt program:

```
#include <iostream.h>

class baza {
public:
    baza() {cout << "Construieste baza\n";}
    ~baza() {cout << "Distruge baza\n";
};

class derivat: public baza {
public:
    derivat() {cout << "Construieste derivat\n";
    ~derivat() {cout << "Distruge derivat\n";}
};
```



```
main()
{
    derivat ob;
    // nu face nimic, decit sa construiasca si sa distruga ob
    return 0;
}
```

După cum precizează comentariul din **main()**, acest program pur şi simplu construieşte şi distruge un obiect numit **ob** care este de clasa **derivat**. Când se execută, acest program afişează:

```
Construieste baza
Construieste derivat
Distruge derivat
Distruge baza
```

După cum puteţi vedea, primul este executat constructorul pentru **baza**, urmat de cel pentru **derivat**. Apoi (deoarece **ob** este distrus imediat în acest program), este apelat destructorul lui **derivat**, urmat de cel pentru **baza**.

Rezultatul experimentului anterior poate fi generalizat. Când este creat un obiect al unei clase derivate, dacă pentru clasa de bază există un constructor, el va fi apelat primul, urmat de constructorul clasei derivate. Când este distrus un obiect al clasei derivate, primul este apelat destructorul său, urmat de cel al clasei de bază, dacă există. Altfel spus, funcţiile constructor sunt executate în ordinea derivării, iar funcţiile destructor în ordine inversă derivării.

Dacă vă gândiţi, există un motiv pentru care funcţiile constructor sunt executate în ordinea derivării. Deoarece o clasă de bază nu ştie nimic despre nici o clasă derivată, este necesar ca orice iniţializare să se facă independent şi anterior oricărei cerinţe anterioare de iniţializare efectuate de clasa derivată. De aceea ea trebuie executată prima.

Tot aşa, este logic ca funcţiile destructor să fie executate în ordinea inversă derivării. Deoarece o clasă de bază conţine o clasă derivată, distrugerea obiectului de bază implică distrugerea obiectului derivat. De aceea, destructorul derivat trebuie să fie apelat înainte ca obiectul să fie distrus complet.

În cazurile de moştenire multiplă (atunci când o clasă derivată devine clasă de bază pentru altă clasă derivată), se aplică regula generală: constructorii sunt apelaţi în ordinea derivării, iar destructorii în ordine inversă. De exemplu, acest program

```
#include <iostream.h>

class baza {
public:
```

```
    baza() {cout << "Construieste baza\n";}
    ~baza() {cout << "Distruge baza\n";}
};

class derivat1 : public baza {
public:
    derivat1() {cout << "Construieste derivat1\n";
    ~derivat1() {cout << "Distruge derivat1\n";}
};

class derivat2 : public derivat1 {
public:
    derivat2() {cout << "Construieste derivat2\n";}
    ~derivat2() {cout << "Distruge derivat2\n";}
};

main()
{
    derivat2 ob;

    // construieste si distruge ob

    return 0;
}
```

afişează această ieşire:

```
Construieste baza
Construieste derivat1
Construieste derivat2
Distruge derivat2
Distruge derivat1
Distruge baza
```

Aceeaşi regulă generală se aplică în situaţii în care apar clase de bază multiple. De exemplu, acest program

```
#include <iostream.h>

class baza1 {
public:
    baza1() {cout << "Construieste baza1\n";}
    ~baza1() {cout << "Distruge baza1\n";}
```



```
};

class baza2 {
public:
    baza2() {cout << "Construieste baza2\n";}
    ~baza2() {cout << "Distruge baza2\n";}
};
class derivat : public baza1, public baza2 {
public:
    derivat() {cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
};

main()
{
    derivat ob;

    // construieste si distruge ob

    return 0;
}
```

afişează această ieşire:

```
Construieste baza1
Construieste baza2
Construieste derivat
Distruge derivat
Distruge baza2
Distruge baza1
```

După cum puteţi vedea, constructorii sunt apelaţi în ordinea derivării - de la stânga la dreapta - după cum s-a specificat în lista de moştenire a clasei **derivat.** Destructorii sunt apelaţi în ordine inversă - de la dreapta la stânga. Aceasta înseamnă că, având specificată **baza2** înaintea clasei **baza1** în lista pentru **derivat**, aşa cum se arată aici:

```
class derivat: public baza2, public baza1 {
```

ieşirea acestui program va arăta astfel:

```
Construieste baza2
Construieste baza1
```

```
Construieste derivat
Distruge derivat
Distruge baza1
Distruge baza2
```

Transmiterea parametrilor spre constructorii clasei de bază

Până acum nici unul dintre exemplele anterioare nu a inclus funcţii constructor cu argumente. În cazurile în care doar constructorul clasei derivate necesită unul sau mai mulţi parametri, folosiţi pur şi simplu sintaxa constructorului parametrizat standard (vedeţi **Capitolul 12**). Totuşi, cum transmiteţi argumentele unei funcţii constructor din clasa de bază? Răspunsul este să folosiţi o formă extinsă a declaraţiei constructorului clasei derivate, care transmite argumentele unuia sau mai multor constructori din clasa de bază. Iată forma generală a declaraţiei extinse pentru constructorul din clasa de bază:

constructor-derivat(listă-argumente) : baza1(lista-argumente),
baza2(lista-argumente),
.
.
.
bazaN(lista-argumente)
{
// corpul constructorului derivat
}

Aici, *baza1* până la *bazaN* sunt numele claselor de bază moştenite de clasa derivată. Reţineţi că declaraţia funcţiei constructor a clasei derivate este separată de clasele de bază prin două puncte iar clasele de bază multiple sunt separate unele de celelalte prin virgule. Să luăm acest program:

```
#include <iostream.h>

class baza {
protected:
    int i;
public:
    baza(int x) {i=x; cout << "Construieste baza\n";}
    ~baza() {cout << "Distruge baza\n";}
};

class derivat: public baza {
```



```
    int j;
public:
    // derivat foloseste x; y este pasat bazei.
    derivat(int x, int y) : baza(y)
          {j=x; cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
    void arata() {cout << i << " " << j << "\n";}
};

main()
{
derivat ob(3, 4);

ob.arata(); // afiseaza 4 3

return 0;
}
```

Constructorul lui **derivat** este declarat aici ca preluând doi parametri, **x** şi **y**. Dar, **derivat()** foloseşte doar **x**; y este pasat către **baza()**. În general, constructorul clasei derivate trebuie să declare atât parametrul/parametrii care este/sunt cerut/ceruţi de el, cât şi pe cel/cei ai clasei de bază. După cum ilustrează exemplul, toţi parametrii ceruţi de clasa de bază îi sunt transmişi acesteia prin constructorul clasei de bază specificate după două puncte.

Iată un exemplu care foloseşte clasele de bază multiple:

```
#include <iostream.h>

class baza1 {
protected:
    int i;
public:
    baza1(int x) {i=x; cout << "Construieste baza1\n";}
    ~baza1() {cout << "Distruge baza1\n";}
};

class baza2 {
protected:
    int k;
public:
    baza2(int x) {k=x; cout << "Construieste baza2\n";}
    ~baza2() {cout << "Distruge baza2\n";}
};
```

```
class derivat: public bazal, public baza2 {
    int j;
public:
    derivat(int x, int y, int z) : bazal(y), baza2(z)
        {j=x; cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
    void arata() {cout << i << " " << j << " " << k << "\n";}
};

main()
{
derivat ob(3, 4, 5);

ob.arata(); // afiseaza 4 3 5

return 0;
}
```

Este important să înţelegeţi că argumentele unui constructor al clasei de bază sunt transmise prin intermediul argumentelor constructorului clasei derivate. De aceea, chiar în cazul în care constructorul unei clase derivate nu foloseşte nici un argument, va fi necesar să declare unul sau mai multe dacă clasa de bază preia unul sau mai multe argumente. În această situaţie, argumentele transmise clasei derivate sunt pur şi simplu transferate bazei. De exemplu, în programul următor, constructorul clasei derivate nu preia nici un argument, dar **baza1()** şi **baza2()** preiau.

```
#include <iostream.h>

class bazal {
protected:
    int i;
public:
    bazal(int x) {i=x; cout << "Construieste bazal\n";}
    ~bazal() {cout << "Distruge bazal\n";}
};

class baza2 {
protected:
    int k;
public:
    baza2(int x) {k=x; cout << "Constriieste baza2\n";}
    ~baza2() {cout << "Distruge baza2\n";}
};

class derivat: public bazal, public baza2 {
public:
    /* Constructorul derivat nu foloseste parametri,
       dar trebuie totusi declarat ca preluandu-i
       pentru a-i pasa claselor de baza.
    */
    derivat(int x, int y) : bazal(y), baza2(y)
        {cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
    void arata() {cout << i << " " << k << "\n";}
};

main()
{
derivat ob(3, 4);

ob.arata(); // afiseaza 3 4

return 0;
}
```


O funcţie constructor a clasei derivate este liberă să folosească oricare dintre parametrii pe care declară că îi preia, chia dacă unul sau mai mulţi sunt transmişi către o clasă de bază. Altfel spus, un argument care este transmis unei clase de bază nu exclude utilizarea sa şi de către o clasă derivată. De exemplu, acest fragment este perfect valid:

```
class derivat: public baza {
    int j;
public:
    // derivat foloseste atit x cit si y si apoi le paseaza
    // bazei.
    derivat(int x, int y): baza(x, y)
        {j = x*y; cout << "Construieste derivat\n";}
```

Încă un lucru de ţinut minte când transmiteţi argumente către constructorii din clasa de bază: argumentele pot să fie orice expresie validă în acel moment, inclusiv apelări de funcţii şi de variabile. Acest lucru este un aspect al faptului că C++ permite iniţializare dinamică.

Permiterea accesului

Când o clasă de bază este moştenită ca **private**, toţi membrii publici şi protejaţi ai acelei clase devin membri privaţi ai clasei derivate. Totuşi, în anumite circumstanţe, va trebui să refaceţi specificatorul de acces original al unuia sau mai multor membri moşteniţi. De exemplu, veţi dori să oferiţi statut public în cadrul clasei derivate anumitor membri publici ai clasei de bază, chiar dacă ea a fost moştenită ca **private**. Pentru a face acest lucru trebuie să folosiţi o *declarare de acces* în interiorul clasei derivate. O declarare de acces are forma generală:

clasa-de-baza::membru;

Declararea de acces este pusă în antetul de acces corespunzător din declararea clasei derivate. Reţineţi că într-o declarare de acces nu este necesară (şi nici permisă) o declarare de tip.

Pentru a vedea cum lucrează o declarare de acces, să începem cu acest scurt fragment:

```
class baza {
public:
    int j; // public in baza
};

// Mosteneste baza ca particular.
class derivat: private baza {
public:
    // aici se afla declararea accesului
    baza::j; // face j din nou public
    .
    .
    .
};
```

După cum ştiţi, deoarece **baza** este moştenită ca **private** în **derivat**, variabila publică **j** este făcută implicit variabilă particulară în **derivat**. Dar, dacă

```
baza::j;
```

este inclusă ca declaraţie de acces în antetul **public** al lui **derivat**, **j** este refăcut cu statutul său **public**.

Pentru a reface dreptul de acces la membrii publici şi protejaţi, puteţi folosi o declarare de acces. Însă, nu puteţi folosi o asemenea declarare pentru a creşte sau scădea statutul de acces la un membru. De exemplu, un membru declarat ca particular într-o clasă de bază nu poate fi făcut public într-o clasă derivată. (Dacă C++ ar permite să se întâmple aceasta, ar distruge mecanismul de încapsulare!)

Următorul program ilustrează declaraţia de acces.

```
#include <iostream.h>

class baza {
    int i; // particular in baza
public:
    int j, k;
    void punei(int x) {i = x;}
    int dai() {return i;}
};

// Mosteneste baza ca particular.
class derivat: private baza {
public:
    /* Urmatoarele trei instructiuni incalca
       mostenirea bazei ca particular si refac
       accesul public la j, punei() si dai(). */
    baza::j; // face din nou j public - dar nu si pe k
    baza::punei; // face public pe punei()
    baza::dai; // face public pe dai()

// baza::i; // ilegal, nu puteti modifica accesul original
    int a; // public
};

main()
{
    derivat ob;
//  ob.i = 10; // ilegal deoarece i este particular in derivat

    ob.j = 20; // legal deoarece j este facut public in derivat
//  ob.k = 30; // ilegal deoarece j este particular in derivat

    ob.a = 40 ; // legal deoarece a este public in derivat
    ob.punei(10);

    cout << ob.dai() << " " << ob.j << " " << ob.a;

    return 0;
}
```

Rețineți cum folosește programul declarațiile de acces pentru a reface statutul public pentru **j**, **punei()** și **dai()**.

Declarațiile de acces sunt acceptate în C++ pentru a se adapta acelor situații în care majoritatea claselor moștenite sunt destinate să fie particulare, dar câțiva membri trebuie să își păstreze statutul lor public sau protejat.

NOTĂ: Deși propunerea de standard ANSI C++ admite încă declarațiile de acces, el descurajează utilizarea lor. Aceasta înseamnă că, deși acesta încă le mai acceptă, este posibil să nu o mai facă în următoarele sale versiuni. În schimb, standardul sugerează realizarea aceluiași efect aplicând cuvântul cheie ***using***. *(Instrucțiunea* ***using*** *va fi discutată mai departe.) Totuși, acum, când scriu această carte, declarațiile de acces sunt încă larg utilizate și nici un compilator folosit uzual nu acceptă cuvântul cheie* ***using***.

Clase de bază virtuale

Când sunt moștenite clase de bază multiple, într-un program în C++ se poate introduce un element de ambiguitate. De exemplu, să luăm acest program incorect:

```
// Acest program contine o eroare si nu va fi compilat.
#include <iostream.h>

class baza {
public:
   int i;
};

// derivat1 mosteneste baza.
class derivat1 : public baza {
public:
   int j;
};

// derivat2 mosteneste baza.
class derivat2 : public baza {
public:
   int k;
};

/* derivat3 mosteneste atat derivat1 cat si derivat2.
   Aceasta inseamna ca in derivat3 exista doua
   copii ale bazei! */
```



```
class derivat3 : public derivat1, public derivat2 {
public:
   int sum;
};

main(void)
{
   derivat3 ob;

   ob.i = 10; // este ambiguu, care dintre i???
   ob.j = 20;
   ob.k = 30;

   // i este aici, de asemenea, ambiguu
   ob.sum = ob.i + ob.j + ob.k;

   // de asemenea ambiguu, care dintre i?
   cout << ob.i << " ";

   cout << ob.j << " " << ob.k << " ";
   cout << ob.sum;

   return 0;
}
```

După cum indică și comentariile din program, atât **derivat1** cât și **derivat2** moștenesc **baza**. Dar, **derivat3** moștenește atât **derivat1** cât și **derivat2**. Aceasta înseamnă că într-un obiect de tipul **derivat3** sunt prezente două copii pentru **baza**. De aceea, pentru o expresie ca

```
ob.i = 20;
```

nu se poate stabili despre care **i** este vorba - cel din **derivat1** sau cel din **derivat2**? Deoarece există două copii ale clasei **baza** în obiectul **ob**, există două **ob.i**.! După cum puteți vedea, instrucțiunea este inerent ambiguă.

Există două căi de remediere a acestui program. Prima este să îi aplicați lui **i** operatorul de specificare a domeniului și să îl selectați manual. De exemplu, această versiune a programului va fi compilată și va rula conform așteptărilor:

```
// Acest program foloseste explicit operatorul de declarare
// a domeniului pentru a-l selecta pe i.
#include <iostream.h>
```

```
class baza {
public:
    int i;
};

// derivat1 mosteneste baza.
class derivat1 : public baza {
public:
    int j;
};

// derivat2 mosteneste baza.
class derivat2 : public baza {
public:
    int k;
};

/* derivat3 mosteneste atat derivat1 cat si derivat2.
   Aceasta inseamna ca in derivat3 exista doua
   copii ale bazei! */
class derivat3 : public derivat1, public derivat2 {
public:
    int sum;
};

main(void)
{
    derivat3 ob;
    ob.derivat1::i = 10; // s-a rezolvat, foloseste i din
    //                      derivat1
    ob.j = 20;
    ob.k = 30;

    // s-a rezolvat
    ob.sum = ob.derivat1::i + ob.j + ob.k;

    // si aici s-a rezolvat
    cout << ob.derivat1::i << " ";

    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;

    return 0;
}
```


După cum puteţi vedea, deoarece s-a aplicat ::, acest program a selectat versiunea **derivat1** din **baza**. Totuşi, această soluţie ridică o problemă mai amplă: ce se întâmplă dacă este cerută efectiv doar o copie pentru **baza**? Există vreo cale de a preveni includerea a două copii în **derivat3**? Răspunsul, după cum probabil aţi ghicit, este da. Soluţia se obţine folosind clasele de bază **virtuale**.

Când două sau mai multe obiecte sunt derivate dintr-o clasă de bază comună, puteţi preveni prezenţa într-unul dintre acestea a mai multor copii ale clasei de bază, declarând-o **virtual** în momentul în care ea este moştenită. Obţineţi aceasta precedând numele clasei de bază cu cuvântul cheie **virtual**. Iată, de exemplu, o altă versiune a programului, exemplu în care **derivat3** conţine doar o singură copie din **baza**:

```
// Acest program foloseste clase de baza virtuale.
#include <iostream.h>

class baza {
public:
    int i;
};

// derivat1 mosteneste baza ca virtal.
class derivat1 : virtual public baza {
public:
    int j;
};

// derivat2 mosteneste baza ca virtual.
class derivat2 : virtual public baza {
public:
    int k;
};

/* derivat3 mosteneste atat derivat1 cat si derivat2.
   De aceasta data exista doar o singura copie a clasei de
   baza. */
class derivat3 : public derivat1, public derivat2 {
public:
    int sum;
};

main(void)
{
    derivat3 ob;
```

```
    ob.i = 10; // acum nu este ambiguu
    ob.j = 20;
    ob.k = 30;

    // nu este ambiguu
    ob.sum = ob.i + ob.j + ob.k;

    // nu este ambiguu
    cout << ob.i << " ";

    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;

    return 0;
}
```

După cum puteţi vedea, cuvântul cheie **virtual** precede restul specificaţiei de acces a clasei moştenite. Acum, deoarece atât **derivat1** cât şi **derivat2** au moştenit **baza ca virtual**, orice moştenire multiplă care le implică va determina prezenţa unei singure copii. De aceea, în **derivat3** există doar o singură copie din **baza**, iar **ob.i = 10** este perfect validă şi nu este ambiguă.

Încă ceva de ţinut minte: chiar dacă atât **derivat1** cât şi **derivat2** specifică **baza** ca fiind **virtual**, **baza** este încă prezentă în obiectele de acel tip. De exemplu, următoarea secvenţă este perfect validă:

```
// defineste o clasa de tipul derivat1
derivat1 clasamea;

clasamea.i = 88;
```

Singura diferenţă dintre o clasă normală şi una virtuală este atunci când un obiect moşteneşte baza mai mult decât o singură dată. Dacă sunt folosite clase de bază virtuale, atunci o singură clasă de bază este prezentă în obiect. Altfel, vor exista copii multiple.

Capitolul 16

Funcţii virtuale şi polimorfism

Polimorfismul (o interfaţă, metode multiple) este admis de C++ atât în timpul compilării cât şi în timpul rulării. Polimorfismul din timpul compilării, obţinut prin funcţiile şi operatorii supraîncărcaţi, a fost discutat în **Capitolul 14.** Polimorfismul din timpul rulării este realizat folosind moştenirea şi funcţiile virtuale, acestea fiind, de altfel, subiectele acestui capitol.

Funcţiile virtuale

O *funcţie virtuală* este o funcţie care este declarată ca fiind **virtual** în clasa de bază şi redefinită de o clasă derivată. Pentru a declara o funcţie ca fiind virtuală, declararea sa este precedată de cuvântul cheie **virtual**. Redefinirea funcţiei în clasa derivată modifică şi are prioritate faţă de definiţia funcţiei din clasa de bază. În esenţă, o funcţie virtuală declarată în clasa de bază acţionează ca un substitut pentru păstrarea datelor care specifică o clasă generală de acţiuni şi declară forma interfeţei. Redefinirea unei funcţii virtuale într-o clasă derivată oferă operaţiile efective pe care le execută funcţia. Altfel spus, o funcţie virtuală defineşte o clasă generală de acţiuni. Redefinirea ei introduce o metodă specifică.

Când sunt utilizate „normal", funcţiile virtuale se comportă exact ca oricare altă funcţie membru al clasei. Însă, ceea ce le face importante şi capabile să admită polimorfismul este modul în care se comportă când sunt apelate printr-un pointer. După cum s-a discutat în **Capitolul 13**, un pointer al clasei de bază poate fi folosit pentru a indica spre orice clasă derivată din acea bază. Când un astfel de pointer indică spre un obiect derivat care conţine o funcţie virtuală, C++ determină care dintre versiunile funcţiei să fie apelată, în funcţie de *tipul obiectului spre care indică* acel pointer. Astfel, când sunt indicate obiecte diferite, sunt executate diferite versiuni ale funcţiei virtuale.

Înainte de a mai discuta teoretic, să examinăm acest scurt exemplu:

```
#include <iostream.h>

class baza {
public:
    virtual void vfunc() {
          cout << "Aceasta este vfunc() din baza.\n";
    }
};
class derivat1 : public baza {
public:
    void vfunc() {
          cout << "Aceasta este vfunc() din derivat1.\n";
    }
};
```



```
class derivat2 : public baza {
public:
    void vfunc() {
          cout << "Aceasta este vfunc() din derivat2.\n";
    }
};

main()
{
    baza *p, b;
    derivat1 d1;
    derivat2 d2;

// indica spre baza
p = &b;
p->vfunc(); // acces la vfunc() din baza

// indica spre derivat1
p = &d1;
p->vfunc(); // acces la vfunc() din derivat1

// indica spre derivat2
p = &d2;
p->vfunc(); // acces la vfunc() din derivat2

return 0;
}
```

Acest program afişează următoarele:

```
Aceasta este vfunc() din baza
Aceasta este vfunc() din derivat1
Aceasta este vfunc() din derivat2
```

După cum ilustrează programul, funcţia **vfunc()** este declarată în interiorul clasei **baza**. Observaţi cum cuvântul cheie **virtual** precede restul declarării funcţiei. Când **vfunc()** este redefinit în **derivat1** şi în **derivat2**, **virtual** nu mai este necesar. (Totuşi, nu este o eroare să îl includeţi atunci când redefiniţi o funcţie virtuală în interiorul unei clase derivate.)

În acest program **baza** este moştenită atât de **derivat1** cât şi de **derivat2**. În interiorul fiecărei definiţii de clasă, **vfunc()** este redefinită relativ la acea clasă. În interiorul funcţiei **main()**, sunt declarate patru variabile:

Nume	Tip
p	Pointer al clasei de baza
b	Obiect din bază
d1	Obiect din derivat1
d2	Obiect din derivat2

Apoi, lui **p** îi este atribuită adresa lui **b**, iar **vfunc()** este apelat prin **p**. Deoarece **p** indică spre un obiect de tipul **baza**, este executată acea versiune a lui **vfunc()**. Apoi, lui **p** i se dă adresa lui **d1** şi, din nou, **vfunc()** este apelată folosind **p**. De această dată, **p** indică spre un obiect de tipul **derivat1**. Aceasta face să fie executată **derivat1::vfunc()**. În sfârşit, lui **p** îi este atribuită adresa lui **d2** şi, din nou, este apelat **vfunc()** iar **p->vfunc()** determină executarea versiunii **vfunc()** din **derivat2**. Esenţialul aici este că versiunea de **vfunc()** ce se va executa este stabilită de tipul de obiect spre care indică **p**. Mai mult, această determinare are loc în timpul rulării, iar procesul stă la baza polimorfismului din timpul rulării.

Deşi puteţi apela o funcţie virtuală în modul „obişnuit", folosind un nume de obiect şi operatorul punct, polimorfismul din timpul rulării este permis doar dacă accesul se face printr-un pointer al clasei de bază. De exemplu, bazându-ne pe programul precedent, următoarea instrucţiune este sintactic validă:

```
d2.vfunc(); // apeleaza vfunc() din derivat2
```

Dacă apelaţi o funcţie virtuală în acest fel, nu faceţi o greşeală, însă nici nu profitaţi de avantajul naturii virtuale a funcţiei **vfunc()**.

La prima vedere, redefinirea unei funcţii virtuale într-o clasă derivată pare similară cu supraîncărcarea funcţiei (overload). Totuşi, nu este aşa, iar operandul *supraîncărcare* nu se aplică redefinirii funcţiilor virtuale deoarece există mai multe diferenţe. Probabil că cea mai importantă este aceea că prototipul pentru o funcţie virtuală redefinită trebuie să coincidă cu cel specificat în clasa de bază. Aceasta diferă de supraîncărcarea unei funcţii normale, pentru care tipurile returnate şi numărul şi tipul parametrilor pot să difere. (De fapt, când supraîncărcaţi o funcţie, ori numărul, ori tipul parametrilor chiar *trebuie* să difere! Prin aceste diferenţe C++ poate să selecteze versiunea corectă a unei funcţii supraîncărcate.) Însă, când este redefinită o funcţie virtuală, toate caracteristicile prototipului său trebuie să fie aceleaşi. Dacă modificaţi prototipul atunci când încercaţi să redefiniţi o funcţie virtuală, compilatorul de C++ o va considera o simplă funcţie supraîncărcată, iar natura sa virtuală se va pierde. O altă restricţie importantă este aceea că funcţiile virtuale nu trebuie să fie membri de tip **static** ai clasei din care fac parte. Mai ales, ele nu pot fi **friend**. În sfârşit, funcţiile constructor nu pot fi virtuale, în schimb cele destructor pot fi.

Datorită restricţiilor şi diferenţelor dintre funcţiile supraîncărcate şi cele virtuale, pentru descrierea redefinirii funcţiei virtuale într-o clasă derivată se foloseşte operandul *suprascriere*.

NOTĂ: O clasă care include o funcţie virtuală este numită o clasă polimorfică.

Atributul virtual este moştenit

Când o funcţie virtuală este moştenită, se moşteneşte şi natura sa virtuală. Aceasta înseamnă că, atunci când o clasă derivată care a moştenit o funcţie virtuală este, ea însăşi, folosită ca o clasă de bază pentru o altă clasă derivată, funcţia virtuală poate fi în continuare suprascrisă. Altfel spus, o funcţie rămâne virtuală indiferent de câte ori este moştenită. De exemplu, să considerăm o versiune a programului precedent:

```
#include <iostream.h>

class baza {
public:
   virtual void vfunc() {
         cout << "Aceasta este vfunc() din baza.\n";
   }

};
class derivat1 : public baza {
public:
   void vfunc() {
         cout << "Aceata este vfunc() din derivat1.\n";
   }
};

/* derivat2 mosteneste functia virtuala vfunc()
   de la derivat1. */
class derivat2 : public derivat1 {
public:
   // vfunc() este inca virtuala
   void vfunc() {
         cout << "Aceasta este vfunc() din derivat2.\n";
   }
};

main()
{
   baza *p, b;
   derivat1 d1;
```

```
    derivat2 d2;

  // indica spre baza
  p = &b;
  p->vfunc(); // acces la vfunc() din baza

  // indica spre derivat1
  p = &d1;
  p->vfunc(); // acces la vfunc() din derivat1

  // indica spre derivat2
  p = &d2;
  p->vfunc(); // acces la vfunc() din derivat2

  return 0;
}
```

După cum este de aşteptat, programul anterior afişează această ieşire:

```
Aceasta este vfunc() din baza
Aceasta este vfunc() din derivat1
Aceasta este vfunc() din derivat2
```

Funcţiile virtuale sunt ierarhizate

După cum ştiţi, când o funcţie este declarată ca fiind **virtual** într-o clasă de bază, ea poate fi suprascrisă de o clasă derivată. Totuşi, funcţia nu trebuie neapărat să fie suprascrisă. Dacă o clasă derivată nu suprascrie funcţia virtuală, atunci, când un obiect din acea clasă derivată are acces la funcţie, este folosită funcţia definită de clasa de bază. Să luăm, de exemplu, acest program:

```
#include <iostream.h>

class baza {
public:
    virtual void vfunc() {
          cout << "Aceasta este vfunc() din baza.\n";
    }

};
class derivat1 : public baza {
public:
```



```
    void vfunc() {
          cout << "Aceata este vfunc() din derivat1.\n";
    }
};

class derivat2 : public baza {
public:
// vfunc() nu este suprascrisa de derivat2, este folosita
// definitia din baza
};

main()
{
    baza *p, b;
    derivat1 d1;
    derivat2 d2;

    // indica spre baza
    p = &b;
    p->vfunc(); // acces la vfunc() din baza

    // indica spre derivat1
    p = &d1;
    p->vfunc(); // acces la vfunc() din derivat1

    // indica spre derivat2
    p = &d2;
    p->vfunc(); // foloseste vfunc() din baza

    return 0;
}
```

Programul precedent produce această ieşire:

```
Aceasta este vfunc() din baza.
Aceasta este vfunc() din derivat1.
Aceasta este vfunc() din baza.
```

Deoarece **derivat2** nu suprascrie **vfunc()**, atunci când **vfunc()** este apelată pentru obiecte de tipul **derivat2,** este folosită funcţia definită de **baza.**

Programul precedent ilustrează un caz special al unei reguli mai generale. Deoarece în C++ moştenirea este ierarhizată, este normal ca funcţiile virtuale să fie, de asemenea, ierarhizate. Aceasta înseamnă că, atunci când o clasă derivată

nu suprascrie o funcție virtuală, este folosită prima redefinire găsită în ordinea inversă a derivării. De exemplu, în următorul program, **derivat2** este derivat din **derivat1**, care este derivat din **baza**. **derivat2** nu suprascrie **vfunc()**. Cum pentru **derivat2** cea mai apropiată versiune de **vfunc()** este cea din **derivat1**, atunci când un obiect din **derivat2** apelează **vfunc()** este folosită **derivat1::vfunc()**.

```
#include <iostream.h>

class baza {
public:
    virtual void vfunc() {
          cout << "Aceasta este vfunc() din baza.\n";
    }
};

class derivat1 : public baza {
public:
    void vfunc() {
          cout << "Aceasta este vfunc() din derivat1.\n";
    }
};

class derivat2 : public derivat1 {
public:
/* vfunc() nu este suprascrisa de derivat2.
   In acest caz, deoarece derivat2 este derivat
   din derivat1, este folosita vfunc() din derivat1.
*/
};

main()
{
    baza *p, b;
    derivat1 d1;
    derivat2 d2;

// indica spre baza
p = &b;
p->vfunc(); // acces la vfunc() din baza

// indica spre derivat1
p = &d1;
p->vfunc(); // acces la vfunc() din derivat1

// indica spre derivat2
p = &d2;
p->vfunc(); // acces la vfunc() din derivat1

return 0;
}
```


Programul precedent afișează următoarele:

```
Aceasta este vfunc() din baza.
Aceasta este vfunc() din derivat1.
Aceasta este vfunc() din derivat1.
```

Funcții virtuale pure

Așa cum ilustrează programul din exemplul precedent, când o funcție virtuală nu este redefinită de o clasă derivată, va fi folosită versiunea definită în clasa de bază. Însă, în multe situații poate să nu existe nici o definiție semnificativă a unei funcții virtuale în cadrul clasei de bază. De exemplu, o clasă de bază poate să nu fie capabilă să definească suficient un obiect pentru a permite să fie creată o funcție virtuală în acea clasă. Mai mult, în unele situații veți dori să vă asigurați că toate clasele derivate suprascriu o funcție virtuală. Pentru a trata aceste două cazuri, C++ admite funcții virtuale pure.

O *funcție virtuală pură* este o funcție virtuală care nu are definiție în clasa de bază. Pentru a declara o astfel de funcție, folosiți forma generală:

virtual *tip nume-functie(lista-de-parametri)* = 0;

Când o funcție virtuală este construită pură, orice clasă derivată trebuie să-i asigure o definiție. În cazul în care clasa derivată nu suprascrie funcția virtuală pură, va rezulta o eroare în timpul compilării.

Următorul program conține un exemplu simplu de funcție virtuală pură. Tipul de bază, **numar**, conține un întreg numit **val**, funcția **puneval()** și funcția virtuală pură **arata()**. Clasele derivate **tiphex**, **tipdec** și **tipoct** moștenesc **numar** și redefinesc **arata()**, astfel încât ea afișează valoarea lui **val** în fiecare bază de numerație (hexazecimală, zecimală și respectiv octală).

```
#include <iostream.h>

class numar {
protected:
```

```
   int val;
public:
   void puneval(int i) {val = i;}

   // arata() este o functie virtuala pura
   virtual void arata() = 0;
};

class tiphex : public numar {
public:
   void arata() {
         cout << hex << val << "\n";
   }
};

class tipdec : public numar {
public:
   void arata() {
         cout << val << "\n";
   }
};

class tipoct : public numar {
public:
   void arata() {
         cout << oct << val << "\n";
   }
};

main()
{
   tipdec d;
   tiphex h;
   tipoct o;

 d.puneval(20);
 d.arata();           // afiseaza 20 - zecimal

 h.puneval(20);
 h.arata();           // afiseaza 14 - hexazecimal

 o.puneval(20);
 o.arata();           // afiseaza 24 - octal
```



```
 return 0;
 }
```

Chiar dacă acest exemplu este foarte simplu, el ilustrează cum o clasă de bază poate să nu fie capabilă să definească semnificativ o funcţie virtuală. În acest caz, numar asigură doar o interfaţă obişnuită pentru tipurile derivate utilizate. De altfel, în acest exemplu, nu există nici un motiv să definim **arata()** în cadrul clasei **numar** deoarece scopul clasei derivate este să afişeze un număr în diferite baze de numeraţie. Desigur, puteţi să creaţi întotdeauna un substitut arbitrar de funcţie virtuală. Însă construind **arata()** ca funcţie pură, ne asigurăm, de asemenea, că toate clasele derivate o vor redefini într-adevăr, pentru a corespunde propriilor lor cerinţe.

Reţineţi că, atunci când o funcţie virtuală este declarată ca fiind pură, toate clasele derivate vor trebui să o suprascrie. Dacă o clasă derivată va omite acest lucru, va rezulta o eroare în timpul compilării.

Clase abstracte

O clasă care conţine cel puţin o funcţie virtuală pură se numeşte *abstractă*. Deoarece o clasă abstractă conţine una sau mai multe funcţii pentru care nu există definiţii (funcţii virtuale pure), nu pot fi create obiecte cu ajutorul ei; o clasă abstractă constituie un tip incomplet care este folosit ca fundament pentru clasele derivate.

Chiar dacă nu puteţi crea obiecte pentru clase abstracte, puteţi crea pointeri şi referinţe spre astfel de clase. Aceasta permite claselor abstracte să admită polimorfismul în timpul rulării, care constă din selectarea funcţiei virtuale corecte de către pointerii clasei de bază.

Utilizarea funcţiilor virtuale

După cum s-a menţionat, unul dintre aspectele centrale ale programării orientate pe obiecte este principiul „o interfaţă, metode multiple". Aceasta înseamnă că poate fi definită o clasă generală de acţiuni pentru care interfaţa este aceeaşi, iar fiecare exemplar (instanţă) specific al clasei generale care defineşte operaţiile efective corespunde propriei sale situaţii specifice. În termenii de C++, o clasă de bază poate fi folosită pentru a defini natura unei interfeţe cu o clasă generală. Fiecare clasă derivată introduce apoi operaţiile specifice cerute de tipurile de date folosite de tipul derivat.

Una dintre cele mai puternice şi mai flexibile căi de introducere a abordării „o interfaţă, metode multiple" este folosirea funcţiilor virtuale, a claselor abstracte şi a polimorfismului din timpul rulării. Folosind aceste caracteristici, creaţi o ierarhizare care trece de la general la specific (de la bază la derivat). Urmând această

filosofie, definiți toate caracteristicile și interfețele comune ale unei clase de bază. În cazurile în care doar clasele derivate pot introduce anumite acțiuni, veți folosi o funcție virtuală pentru a defini interfața care va fi folosită de clasa sau clasele derivate. În esență, în clasa de bază creați și definiți tot ce se referă la cazul general. Clasa derivată completează detaliile specifice.

În continuare este dat un exemplu simplu, care ilustrează valoarea conceptului „o interfață, metode multiple". Pentru a efectua o conversie de unități dintr-un sistem într-altul (de exemplu, litri în galoane) este creată o ierarhie de clase. Clasa de bază **convert** declară două variabile, **val1** și **val2**, care păstrează valorile inițială și respectiv convertită. Ea mai definește funcțiile **dainit()** și **daconv()** care returnează valoarea inițială și cea convertită. Aceste elemente din **convert** sunt fixe și aplicabile tuturor claselor derivate care vor moșteni **convert**. Dar, funcția care va efectua efectiv conversia, **calcul()**, este o funcție virtuală pură care trebuie să fie definită de clasa derivată din **convert**. Natura specifică a funcției **calcul()** va fi determinată de tipul de conversie care are loc.

```
// Exemplu practic de functie virtuala.
#include <iostream.h>

class convert {
protected:
   double val1; // valoare initiala
   double val2; // valoare convertita

public:
   convert(double i) {
         val1 = i;
   }
   double daconv() {return val2;}
   double dainit() {return val1;}

   virtual void calcul() = 0;
};

// Litri in galoane.
class l_in_g : public convert {
public:
   l_in_g(double i) : convert (i) { }
   void calcul() {
         val2 = val1 / 3.7854;
   }
};
```



```
// Fahrenheit in Celsius
class f_in_c : public convert {
public:
   f_in_c(double i) : convert(i) { }
   void calcul() {
         val2 = (val1-32) / 1.8;
   }
};

main()
{
   convert *p; // pointer spre clasa de baza

   l_in_g lgob(4);
   f_in_c fcob(70);

   // pentru conversie foloseste mecanismul functiei virtuale
   p = &lgob;
   cout << p->dainit() << " litri reprezinta ";
   p->calcul();
   cout << p->daconv() << " galoane\n"; // l_in_g
   p = &fcob;
   cout << p->dainit() << " in Fahrenheit reprezinta ";
   p = compute();
   cout << p->daconv() << " Celsius\n"; // f_in_c
   return 0;
}
```

Acest program creează două clase derivate din **convert**, numite **l_in_g** și **f_in_c**. Ele efectuează conversiile pentru litri în galoane și respectiv pentru grade Fahrenheit în grade Celsius. După cum puteți vedea, pentru a efectua conversia dorită, fiecare clasă derivată suprascrie **calcul()** în felul ei propriu. Totuși, deși conversia efectivă (deci, metoda) diferă între **l_in_g** și **f_in_c**, interfața rămâne constantă.

Una dintre calitățile claselor derivate și ale funcțiilor virtuale este aceea că tratarea cazurilor noi se realizează foarte ușor. De exemplu, având programul anterior, puteți să adăugați o conversie de picioare în metri incluzând această clasă:

```
// Picioare in metri
class f_in_m : public convert {
public:
   f_in_m(double i) : convert(i) { }
```

```
    void calcul() {
        val2 = val1 / 3.28;
    }
};
```

O utilizare importantă a claselor abstracte şi a funcţiilor virtuale este cea a *bibliotecilor de clase*. Puteţi crea o bibliotecă de clase generice şi extensibile care vor fi folosite de alţi programatori. Programatorul va moşteni clasa generală creată de dvs., care defineşte interfaţa şi elementele comune, şi va defini doar acele funcţii specifice clasei derivate. Prin bibliotecile de clase sunteţi capabili să creaţi şi să controlaţi interfaţa unei clase generale, permiţând în continuare altor programatori să o adapteze situaţiilor lor specifice.

În încheiere: clasa de bază **convert** este un exemplu de tip incomplet. Funcţia virtuală **calcul()** nu este definită în interiorul clasei **convert** deoarece nu se poate asigura o definire semnificativă. Clasa **convert** pur şi simplu nu conţine suficiente informaţii pentru a se putea defini **calcul()**. Se creează un tip complet doar când **convert** este moştenit de o clasă derivată.

Legături iniţiale / ulterioare

Înainte de a încheia acest capitol despre funcţii virtuale şi polimorfism în timpul rulării, mai există de definit doi operanzi frecvent utilizaţi în discuţiile despre C++ şi programarea orientată pe obiecte. Ei sunt *legături iniţiale* (early binding) şi *legături ulterioare* (late binding).

Legături iniţiale se referă la evenimentele care apar în timpul compilării. În esenţă, ele semnifică faptul că toate informaţiile necesare pentru a apela o funcţie sunt cunoscute în timpul compilării. (Altfel spus, legăturile iniţiale arată că legăturile dintre un obiect şi o apelare de funcţie se realizează în timpul compilării.) Exemplele de legături iniţiale includ apelările de funcţii normale (inclusiv funcţiile standard de bibliotecă), apelări de funcţii supraîncărcate şi operatori de supraîncărcare. Avantajul principal al legăturilor iniţiale este eficienţa. Deoarece toate informaţiile necesare pentru a apela o funcţie sunt determinate în timpul compilării, aceste tipuri de funcţii sunt foarte rapide.

Opusul legăturilor iniţiale sunt *legăturile ulterioare*. În C++, legăturile ulterioare se referă la apelările de funcţii care nu sunt rezolvate până în momentul rulării. Pentru realizarea legăturilor ulterioare sunt folosite funcţiile virtuale. După cum ştiţi, când accesul se face printr-un pointer din bază, apelarea efectivă a funcţiei virtuale este determinată de tipul obiectului spre care indică pointerul. Deoarece în majoritatea cazurilor acesta nu poate fi determinat în timpul compilării, obiectul şi funcţia nu sunt legate până în momentul rulării. Avantajul principal al legăturilor ulterioare este flexibilitatea. Spre deosebire de legăturile iniţiale, cele ulterioare vă permit să creaţi programe care să răspundă la evenimentele ce apar în timpul rulării fără să trebuiască să creaţi o cantitate mare de „cod pentru contingenţe". Reţineţi că, deoarece o apelare de funcţie nu este rezolvată până în momentul rulării, legătura ulterioară va determina timpi de execuţie ceva mai lenţi.

Capitolul 17

Bazele sistemului de I/O din C++

Pe lângă faptul că permite sistemul de I/O din C, C++ defineşte sistemul său propriu orientat pe obiecte. Ca şi sistemul de I/O din C, cel din C++ este complet integrat. Aceasta înseamnă că diferitele aspecte ale sistemului de I/O din C++, cum ar fi I/O de la consolă sau I/O de pe disc, sunt doar perspective diferite ale aceluiaşi mecanism. Acest capitol discută bazele sistemului de I/O orientat pe obiecte din C++. Chiar dacă exemplele din acest capitol folosesc I/O de la „consolă", informaţiile sunt aplicabile şi altor echipamente, inclusiv fişierelor de pe disc (discutate în **Capitolul 18**).

După cum ştiţi, sistemul de I/O din C este foarte bogat, flexibil şi puternic. Vă puteţi întreba de ce C++ introduce un alt sistem. Răspunsul este acela că sistemul de I/O din C nu cunoaşte obiectele. De aceea, pentru ca C++ să asigure un suport complet pentru programarea orientată pe obiecte, a fost necesar să se creeze un sistem de I/O orientat pe obiecte, care să poată opera cu obiectele create de utilizator. În afară de admiterea obiectelor, mai există şi alte câteva efecte secundare benefice ale utilizării sistemului de I/O din C++, chiar şi în programele care nu folosesc extensiv (sau chiar de loc) obiecte definite de utilizator. Veţi vedea câteva exemple mai departe, în acest capitol.

În acest capitol, veţi învăţa despre modul de formatare a datelor. Veţi mai învăţa despre cum se suprapun operatorii din C++ << şi >> astfel încât să poată fi folosiţi împreună cu clasele pe care le creaţi. De asemenea, veţi vedea cum se creează funcţiile speciale de I/O, numite manipulatori, care pot face programul dvs. mai eficient.

Streamuri în C++

Ca şi sistemul de I/O din C, cel din C++ operează prin streamuri. Ele au fost prezentate în detaliu în **Capitolul 9**; discuţia nu se va mai relua aici. Totuşi, să rezumăm: un stream este o entitate logică ce produce sau primeşte informaţie. Un stream este legat de un echipament fizic prin sistemul de I/O din C++. Toate streamurile se comportă în acelaşi fel, chiar dacă echipamentele fizice la care sunt conectate efectiv pot să difere substanţial. Deoarece toate streamurile se comportă la fel, aceleaşi funcţii de I/O din C++ pot opera, teoretic, asupra oricărui tip de echipament fizic. De exemplu, puteţi folosi aceeaşi funcţie care scrie într-un fişier şi pentru a scrie la imprimantă sau pe ecran. Avantajul acestei facilităţi este că trebuie să învăţaţi doar o singură interfaţă.

Clasele de bază pentru streamuri

C++ asigură suportul pentru sistemul său de I/O în fişierul antet IOSTREAM.H.Aici sunt definite două ierarhii de clase care admit operaţii de I/O. Clasa cu nivelul cel mai mic se numeşte **streambuf** şi asigură operaţiile de bază de intrare şi de ieşire. Nu veţi folosi **streambuf** direct, decât dacă veţi deriva propriile clase de I/O. A doua ierarhie porneşte cu clasa **ios**, care acceptă I/O formatate. Din ea sunt

derivate clasele **istream**, **ostream** şi **iostream**. Aceste clase sunt folosite pentru a crea streamuri capabile să introducă, să obţină şi respectiv să introducă/să obţină. După cum veţi vedea în capitolele următoare, din **ios** sunt derivate multe alte clase pentru utilizarea fişierelor de pe disc şi formatarea în RAM.

Clasa **ios** conţine multe funcţii membre şi variabile membre care controlează sau urmăresc operaţiile fundamentale ale streamului. În cursul acestui capitol şi al următorului, se vor face multe referiri la membrii săi. Reţineţi doar că dacă folosiţi sistemul de I/O din C++ în manieră normală, membrii clasei **ios** vor fi capabili să lucreze cu orice stream.

Streamuri predefinite în C++

Când îşi începe execuţia un program în C++, se deschid automat patru streamuri încorporate. Ele sunt

Stream	Semnificaţie	Echipament implicit
cin	Intrare standard	Tastatură
cout	Ieşire standard	Ecran
cerr	Ieşire standard pentru eroare	Ecran
clog	Versiune cu memorie tampon pentru cerr	Ecran

Streamurile **cin**, **cout** şi **cerr** corespund streamurilor **stdin**, **stdout** şi **stderr** din C.

Implicit, streamurile standard sunt folosite pentru a comunica cu consola. Însă, în mediile care admit redirecţionarea I/O (cum ar fi DOS, Unix, OS/2 şi Windows), streamurile standard pot fi redirecţionate spre alte echipamente sau fişiere. Totuşi, pentru simplicitate, exemplele din acest capitol presupun că nu apare nici o redirecţionare I/O.

NOTĂ: Standardul propus ANSI C++ mai defineşte următoarele patru streamuri: ***win, wout, werr*** *şi* ***wlog****. Ele sunt versiunile streamurilor standard pentru caractere mari (wide caracters). Acestea sunt de tipul* ***wchar_t*** *şi, în general, au mărimea de 16 biţi; ele sunt folosite pentru seturile lărgite de caractere necesare anumitor limbi.*

I/O formatate

Sistemul de I/O din C++ vă permite să formataţi operaţiile de I/O. De exemplu, puteţi să daţi mărimea unui câmp, să specificaţi baza unui număr sau să determinaţi câte cifre se vor afişa după punctul zecimal. În esenţă, orice format pe care puteţi să îl obţineţi sau să-l introduceţi cu funcţiile din C **printf()** şi **scanf()** poate fi, de asemenea, obţinut sau introdus folosind operatorii de I/O din C++, << şi >>.

Există două căi înrudite, dar conceptual diferite, prin care puteţi să formataţi datele. În primul rând, puteţi avea acces direct la diferiţi membri ai clasei **ios**. Mai concret, puteţi să controlaţi diverşi indicatori pentru format, definiţi în cadrul clasei **ios**, sau să apelaţi diferite funcţii membre din **ios**. În al doilea rând, puteţi folosi funcţii speciale numite *manipulatori*, care pot fi incluse în expresii de I/O.

Vom începe discuţia despre I/O formatate folosind funcţiile membre, **ios** şi indicatorii.

Formatarea folosind membrii ios

Fiecărui stream îi este asociat un set de indicatori pentru format care controlează unele dintre căile prin care sunt formatate informaţiile de către un stream. În **ios**, în mod normal, indicatorilor li se atribuie nume şi valoare prin enumerare, aşa cum se arată în următorul exemplu:

```
// indicatori de formatare din ios
enum {
    skipws = 0x0001,
    left = 0x0002,
    right = 0x0004,
    internal = 0x0008,
    dec = 0x0010,
    oct = 0x0020,
    hex = 0x0040,
    showbasa = 0x0080,
    showpoint = 0x0100,
    uppercase = 0x0200,
    showpos = 0x0400,
    scientific = 0x800,
    fixed = 0x1000,
    unitbuf = 0x2000,
};
```

Indicatorii de format asociaţi cu un stream sunt codificaţi sub o anumită formă de întregi lungi. Standardul propus pentru ANSI C++ specifică tipul indicatorilor de format ca fiind **fmtflags**. Dar, nici un compilator larg răspândit nu defineşte curent acest tip. (Desigur, toate o vor face în viitorul apropiat.) Din punct de vedere practic, este aproape sigur că **fmtflags** va fi un simplu nume pentru **typedef** dat cu un întreg lung. Această carte va folosi tipul **long** când se va referi la indicatorii de formatare deoarece acesta este tipul folosit curent de toate compilatoarele de C++ uzuale. Va trebui însă să verificaţi manualul compilatorului.

Când este activat indicatorul **skipws**, caracterele de spaţii libere din faţă (spaţii simple, de tabulare şi de linie nouă) sunt eliminate atunci când se efectuează intrarea unui stream. Când **skipws** este şters, caracterele libere nu sunt eliminate.

Când este activat indicatorul **left**, ieşirea este aliniată la stânga. Când este activat **right**, ieşirea este aliniată la dreapta. Când **internal** este activ, o valoare numerică dintr-un câmp este completată cu spaţii inserate pornind de la semn sau de la caracterul bazei. (Veţi învăţa, în curând, cum să specificati mărimea unui câmp.) Dacă nici unul dintre acestea nu este activ, ieşirea este aliniată implicit la dreapta.

Tot implicit valorile numerice sunt obţinute în sistemul zecimal. Dar, puteţi modifica baza numărului. Activarea indicatorului **oct** determină ca ieşirea să fie afişată în octal. Activarea indicatorului **hex** face ca rezultatul să fie afişat în hexazecimal. Pentru a reveni la ieşirea în sistem zecimal, activaţi indicatorul **dec**. Aceste indicatoare determină şi baza când sunt introduse valori întregi.

Activarea indicatorului **showbase** determină afişarea bazei numărului. De exemplu, dacă baza de conversie este în hexazecimal, valoarea F1 va fi afişată ca 0x1F.

Când este afişată notaţia ştiinţifică, „e" este implicit scris cu literă mică. De asemenea, când se afişează o valoare hexazecimală, „x" este cu literă mică. Când este activ **uppercase**, aceste caractere sunt afişate cu literă mare.

Activarea indicatorului **showpos** determină ca în faţa valorilor pozitive să fie afişat un semn plus.

Activarea indicatorului **showpoint** determină, pentru ieşirile în virgulă mobilă, afişarea punctului zecimal şi a zerourilor finale - indiferent dacă sunt semnificative sau nu.

Activarea indicatorului **scientific** determină ca valorile în virgulă mobilă să fie afişate în notaţie ştiinţifică. Când este activat **fixed**, numerele în virgulă mobilă apar în notaţia normală, implicit, cu şase locuri pentru zecimale. Dacă nu este activ nici un indicator, compilatorul alege o metodă corespunzătoare.

Când este activ **unitbuf**, sistemul de I/O din C++ este golit după fiecare operaţie de ieşire.

> ***NOTĂ:*** *Indicatorii de formatare descrişi mai înainte vor fi admişi de orice compilator de C++. Dar, în momentul scrierii acestei cărţi, natura exactă a indicatorilor de formatare din **ios** este încă în curs de definire de către comitetul de standardizare ANSI C++. De exemplu, a fost adăugat numele **boolalpha**, care permite operaţii de I/O asupra noului tip de date definit, **bool**. Acest indicator nu este definit curent de majoritatea compilatoarelor. Verificaţi manualul compilatorului dvs. pentru a vedea dacă sunt accesibile pentru utilizare acesta sau alţi indicatori de format.*

Activarea indicatorilor de format

Pentru a activa un indicator de format, folosiţi funcţia **setf()**. Această funcţie este membru al clasei **ios**. Iată forma sa cea mai uzuală:

long setf(long *indicator*);

Funcţia returnează valorile precedente ale indicatorului de format şi activează acei indicatori specificaţi ca argumente; ceilalţi rămân neschimbaţi.

De exemplu, pentru a activa **showpos** puteţi folosi această instrucţiune:

```
stream.setf(ios::showpos);
```

Aici, *stream* este streamul pe care doriţi să îl modificaţi. De exemplu, următorul program afişează valoarea lui 100 în hexazecimal şi îi indică baza.

```
#include <iostream.h>

main()
{
    cout.setf(ios::hex);
    cout.setf(ios::showbase);

    cout << 100; // afiseaza 0x64

    return 0;
}
```

Este important să înţelegeţi că **setf()** este o funcţie membră a clasei **ios** şi are efect asupra streamurilor create de acea clasă. De aceea, orice apelare pentru **setf()** este făcută relativ la un anumit stream. Se poate invoca **setf()** în abstract. Altfel spus, în C++ nu există ideea de stare de format global. Fiecare stream îşi întreţine propriile informaţii referitoare la starea formatului.

Chiar dacă nu este practic greşit, există o cale mai eficientă de a scrie programul anterior. În loc să apelăm **setf()** de mai multe ori, puteţi să uniţi prin OR (SAU logic) valorile indicatorilor pe care doriţi să-i activaţi. De exemplu, următoarea instrucţiune realizează singură acelaşi lucru.

```
// Puteti uni cu OR doi sau mai multi indicatori
cout.setf(ios::showbasa | ios::hex);
```

*REŢINEŢI: Deoarece indicatorii de format sunt definiţi în interiorul clasei **ios**, pentru a avea acces la valorile lor trebuie să folosiţi **ios** şi operatorul de specificare a domeniului. De exemplu, **showbase** nu va fi recunoscută ca atare. Trebuie să specificaţi **ios::showbase**.*

Dezactivarea indicatorilor de format

Complementul funcţiei **setf()** este **unsetf()**. Această funcţie membru al lui **ios** este folosită pentru a şterge unul sau mai mulţi indicatori de format. Forma sa generală este:

long unsetf(long *indicatori*);

Sunt dezactivaţi indicatorii specificaţi în paranteze. (Toţi ceilalţi rămân neafectaţi.) Se returnează starea anterioară a indicatorilor.

Următorul program ilustrează **unsetf()**. Pentru început, el activează indicatorii **uppercase şi scientific.** Apoi el transpune 100.12 în notaţie ştiinţifică. În acest caz, E folosit la notaţia ştiinţifică este scris mare. După care dezactivează indicatorul **uppercase** şi scrie din nou 100.12 în notaţie ştiinţifică, folosind litera „e".

```
#include <iostream.h>

main()
{
    cout.setf(ios::uppercase | ios::scientific);

    cout << 100.12; // afiseaza 1.0012E+02

    cout.unsetf(ios::uppercase); // dezactiveaza uppercase

    cout << " \n" << 100.12; // afiseaza 1.0012e+02

    return 0;
}
```

O formă suprapusă a funcţiei setf()

Există o formă suprapusă a funcţiei **setf()** care are sintaxa generală:

long setf(long *indicatori1*, long *indicatori2*);

În această versiune, sunt afectaţi doar indicatorii specificaţi de *indicatori2*.. Ei sunt, pentru început, dezactivaţi şi apoi activaţi în funcţie de *indicatori1*. Reţineţi că, şi dacă *indicatori1* conţin şi alţi indicatori care nu sunt specificaţi de *indicatori2*, vor fi modificaţi doar cei ce apar în *indicatori2*. Este returnată starea anterioară a indicatorului. De exemplu, următorul program activează **showpos** şi **showpoint**. Apoi el îi dezactivează pe amândoi şi reactivează **showpos**.

```
#include <iostream.h>

main()
{

   cout.setf(ios::showpos | ios::showpoint);

   cout << 10.00 << " \n"; // afiseaza +10.000000

   cout.setf(ios::showpoint, ios::showpos |
             ios::showpoint);

   cout << 10.00;
   // showpos este dezactivat, se afiseaza 10.00000

   return 0;
}
```

Rețineți că doar indicatorii specificați în *indicatori2* pot fi afectați de indicatorii specificați în *indicatori1*. De exemplu, următorul program nu va lucra.

```
// Acest program nu va lucra.
#include <iostream.h>

main()
{
   cout.setf(ios::showbase | ios::hex);

   cout << 100; // afiseaza 0x64

   cout.setf(ios::oct, ios::hex);
   // eroare, oct nu apare dupa virgula

   cout << " \n" << 100; // afiseaza 100 - nu 0144

   return 0;
}
```

Aici, în apelarea funcției **setf()**, *indicatori2* specifică faptul că doar **hex** poate fi afectat. Deoarece valoarea din *indicatori1* este **oct**, **hex** este dezactivat, dar **oct** nu este activat. Următorul program arată o versiune corectă.


```
// Acest program este acum corect.
#include <iostream.h>

main()
{
   // puteti sa alaturati prin SAU logic doi sau mai multi
   // indicatori
   cout.setf(ios::showbase | ios::hex);

   cout << 100; // afiseaza 0x64

   // acum oct poate fi activat
   cout.setf(ios::oct, ios::hex | ios::oct);

   cout << " \n" << 100; // afiseaza 0144

   return 0;
}
```

Se pot face referiri la oricare dintre câmpurile **oct**, **dec** și **hex** prin numele colectiv **ios::basefield**. Ne putem referi similar și la câmpurile **left**, **right** și **internal** cu **ios::adjustfield**. În sfârșit, ne putem referi la câmpurile **scientific** și **fixed** cu **ios::floatfield**. De exemplu, programul precedent ar putea fi scris astfel:

```
#include <iostream.h>

main()
{
   // puteti sa alaturati prin SAU logic doi sau mai multi
   // indicatori
   cout.setf(ios::showbasa | ios::hex);

   cout << 100; // afiseaza 0x64
   // foloseste ios::basefield
   cout.setf(ios::oct, ios::basefield);

   cout << " \n" << 100; // afiseaza 0144

   return 0;
}
```

Rețineți că, de cele mai multe ori, veți prefera să folosiți **unsetf()** pentru a dezactiva indicatorii și versiunea cu un singur parametru a funcției **setf()**, cea descrisă aici, pentru a-i activa. Versiunea **setf(long** *indicatori1*, **long** *indicatori2*)

este folosită în situații speciale. De exemplu, puteți avea un șablon pentru un indicator care specifică starea tuturor indicatorilor de format, dar doriți să modificați doar unul sau doi. În acest caz, puteți specifica șablonul în *indicatori1* și *indicatori2* pentru a indica pe care dintre aceștia îi veți modifica.

Examinarea indicatorilor de format

Vor fi cazuri când veți dori doar să cunoașteți starea curentă a indicatorilor de format dar să nu modificați nici unul. Pentru a realiza aceasta, **ios** include, de asemenea, funcția membru **flags()**, care returnează pur și simplu starea curentă a fiecărui indicator de format într-un număr întreg lung. Iată prototipul său:

long flags();

Următorul program folosește **flags()** pentru a afișa indicatorii de format relativi la **cout**. Fiți atenți, în special, la funcția **showflags()**. Probabil că o veți găsi folositoare pentru programele pe care le veți scrie.

```
#include <iostream.h>

void showflags();

main()
{
    // arata starea implicita a indicatorilor de format
    showflags();

    cout.setf(ios::right | ios::showpoint | ios::fixed);

    showflags();

    return 0;
}

// Aceasta functie afiseaza starea indicatorilor de format.
void showflags()
{
    long f, i;
    int j;

    char indic[15][12] = {
          "skipws",
          "left",
```



```
          "right",
          "internal",
          "dec",
          "oct",
          "hex",
          "showbase",
          "showpoint",
          "uppercase",
          "showpos",
          "scientific",
          "fixed",
          "unitbuf",
    };

    f = cout.flags(); // obtine starea indicatorilor

    // verifica fiecare indicator
    for(i=1, j=0; i<=0x2000; i = i<<1, j++)
          if(i & f) cout << numgen[j] << " este activat \n";
          else cout << numgen[j] << " este dezactivat \n";

    cout << " \n";
}
```

Iată ieșirea acestui program:

```
skipws este activat
left este dezactivat
right este dezactivat
internal este dezactivat
dec este dezactivat
oct este dezactivat
hex este dezactivat
showbase este dezactivat
showpoint este dezactivat
uppercase este dezactivat
showpos este dezactivat
scientific este dezactivat
fixed este dezactivat
unitbuf este dezactivat

skipws este activat
left este dezactivat
```

```
right este activat
internal este dezactivat
dec este dezactivat
oct este dezactivat
hex este dezactivat
showbase este dezactivat
showpoint este activat
uppercase este dezactivat
showpos este dezactivat
scientific este dezactivat
fixed este activat
unitbuf este dezactivat
```

NOTĂ: De vreme ce este posibil ca un compilator de C++ să definească alte valori pentru indicatorii de format decât cele folosite în programul precedent, puteți să vedeți alte rezultate decât cele prezentate aici. Dacă este așa, verificați în manualul compilatorului dvs. ce valori folosește acesta.

Activarea tuturor indicatorilor

Funcția **flags()** are o a doua formă care vă permite să activați toți indicatorii de format asociați unui stream. Prototipul pentru această versiune a funcției **flags()** este prezentat aici:

long flags(long *f*);

Când folosiți această versiune, modelul de biți aflat în *f* este copiat în variabila folosită pentru a păstra indicatorii pentru format asociați acelui stream. De aceea, sunt afectați toți indicatorii. Funcția returnează starea anterioară.

Următorul program ilustrează această versiune a funcției **flags()**. Pentru început, ea construiește o grilă pentru indicatori care activează **showpos**, **showbase**, **oct** și **right**. Pentru majoritatea compilatoarelor de C++, ei au valorile 0x0400, 0x0080, 0x0020 și 0x0004. Când sunt grupați rezultă valoarea folosită în program, 0x04A4. Toți ceilalți indicatori sunt dezactivați. Apoi, se folosește **flags()** pentru a atribui aceste valori variabilei asociate cu **cout**. Funcția **showflags()** verifică dacă indicatorii au fost activați așa cum s-a indicat. (Este aceeași cu cea din programul anterior.)

```
#include <iostream.h>

void showflags();
```



```
main()
{
    // arata starea implicita a indicatorilor de format
    showflags();

    // showpos, showbase, oct, right sunt activate,
    // ceilalti nu
    long f = 0x04A4;
    cout.flags(f); // activeaza toti indicatorii

    showflags();

    return 0;
}
```

Utilizarea funcțiilor width(), precision() și fill()

În afară de indicatorii de format, în **ios** mai sunt definite trei funcții membre care activează următorii parametri pentru format: mărimea câmpului, precizia și caracterul de umplere. Funcțiile care execută aceste trei lucruri sunt **width()**, **precision()** și respectiv **fill()**. Ele vor fi examinate pe rând.

Implicit, când este obținută o valoare, ea ocupă doar atât spațiu câte caractere îi sunt necesare pentru a o afișa. Dar, puteți specifica o mărime de câmp minim folosind funcția **width()**. Prototipul ei este prezentat aici:

int width(int *w*);

w devine, aici, mărimea câmpului și este returnată mărimea anterioară a acestuia. Pentru unele implementări, mărimea câmpului trebuie specificată înainte de fiecare ieșire. Dacă nu este specificată, este folosită mărimea implicită a câmpului.

După ce ați specificat lungimea minimă a câmpului, când o valoare folosește mai puțin decât această lungime, pentru a se atinge lungimea specificată, câmpul va fi completat cu caracterul de umplere curent (implicit, spațiul). Rețineți însă că, dacă dimensiunea valorii este mai mare decât lungimea minimă a câmpului, câmpul va fi extins. Valorile nu vor fi trunchiate.

Când obțineți valori în virgulă mobilă, puteți determina numărul de cifre care să fie afișate după punctul zecimal folosind funcția **precision()**. Iată prototipul său:

int precision(int *p*);

Aici, precizia este stabilită prin *p* și se returnează valoarea anterioară. Precizia implicită este 6. În unele implementări precizia trebuie specificată înainte de

fiecare ieşire în virgulă mobilă. Dacă nu o specificaţi, este folosită precizia implicită.

Când un câmp trebuie să fie completat, el este umplut, implicit, cu spaţii. Dar, puteţi specifica un caracter de umplere folosind funcţia **fill()**. Prototipul său este:

```
char fill(char ch);
```

După o apelare a funcţiei **fill()**, *ch* devine noul caracter de umplere şi se returnează cel anterior.

Iată un program care ilustrează aceste funcţii:

```
#include <iostream.h>

main()
{
    cout.precision(4);

    cout.width(10);

    cout <, 10.12345 << "\n"; // afiseaza 10.12

    cout.fill('*');

    cout.width(10);
    cout << 10.12345 << "\n"; // afiseaza *****10.12

    // marimea cimpului se aplica, de asemenea, si sirurilor
    cout.width(10);
    cout << "Salut" << "\n"; // afiseaza *****Salut
    cout.width(10);
    cout.setf(ios::left); // aliniere la stanga
    cout << 10.12345; // afiseaza 10.12*****

    return 0;
    }
```

Aici este prezentată ieşirea acestui program:

```
     10.12
*****10.12
*****Salut
10.12*****
```


Utilizarea manipulatorilor pentru I/O formatate

A doua cale prin care puteţi modifica parametrii de formatare ai streamului este aceea de folosire a funcţiilor speciale numite *manipulatori*, care pot fi incluse în expresii de I/O. Manipulatorii standard sunt prezentaţi în **Tabelul 17-1**. După cum puteţi vedea examinând tabelul, mulţi manipulatori de I/O sunt paraleli funcţiilor din clasa **ios**.

*NOTĂ: Manipulatorii prezentaţi în Tabelul 17-1 sunt admişi de toate compilatoarele de C++. Totuşi, în momentul scrierii acestei cărţi, natura exactă a manipulatorilor de I/O este în curs de definire de către comitetul de standardizare ANSI C++. De exemplu, manipulatorul **boolalpha()** a fost adăugat pentru a permite operaţii de I/O pentru **bool**, noul tip de date definit. Acest manipulator s-ar putea să nu fie acceptat de compilatorul dvs.. Verificaţi în manualul compilatorului ce alţi manipulatori sunt disponibili.*

Pentru a avea acces la manipulatorii care preiau parametri (aşa cum este **setw()**), trebuie să includeţi în programul dvs. IOMANIP.H.

Iată un exemplu care foloseşte unii dintre manipulatori:

Manipulator	Scop	Intrare/Ieşire
dec	Intrare/ieşire de date în zecimal	Intrare şi ieşire
endl	Obţine un caracter de linie nouă şi goleşte streamul	Ieşire
ends	Obţine un null	Ieşire
flush	Goleşte un stream	Ieşire
hex	Introduce / obţine date în hexazecimal	Intrare şi ieşire
oct	Introduce / obţine date în octal	Intrare şi ieşire
resetiosflags(long *f*)	Dezactivează indicatorii specificaţi în f	Intrare şi ieşire
setbase(int *baza*)	Stabileşte baza ca *baza* numerică	Ieşire
setfill(int *ch*)	Stabileşte *ch* drept caracter de umplere	Ieşire
setiosflags(long *f*)	Activează indicatorii specificaţi în f	Intrare şi ieşire
setprecision(int *p*)	Stabileşte p ca număr de cifre pentru precizie	Ieşire
setw(int *w*)	Stabileşte *w* ca lungime a câmpului	Ieşire
ws	Omite spaţiile libere de la început	Intrare

Tabelul 17-1 *Manipulatorii în C++*

```
#include <iostream.h>
#include <iomanip.h>

main()
{
    cout << hex << 100 << endl;

    cout << setfill('?') << setw(10) << 2343.0;

    return 0;
}
```

Acesta afişează:

```
64
??????2343
```

Reţineţi cum apar manipulatorii în lanţul de operaţii de I/O. Reţineţi, de asemenea, că, atunci când un manipulator nu preia un argument, aşa cum este **endl()** din exemplu, el nu este urmat de paranteze (din cauză că operatorului supraîncărcat **<<** îi este transmisă adresa funcţiei).

Pentru comparaţie, iată o versiune echivalentă funcţional a programului precedent care foloseşte funcţiile membre din **ios** pentru a obţine aceleaşi rezultate:

```
#include <iostream.h>
#include <iomanip.h>

main()
{
    cout.setf(ios::hex);
    cout << 100 << "\n"; // 100 in hex

    cout.fill('?');
    cout.width(10);
    cout << 2343.0;

    return 0;
}
```

Aşa cum sugerează exemplul, avantajul principal al folosirii manipulatorilor în locul funcţiilor membre ale clasei **ios** este acela că ei vă permit deseori să scrieţi un cod mult mai compact.

Puteţi să folosiţi manipulatorul **setiosflags()** pentru a activa direct diverşii indicatori pentru format referitori la un stream. De exemplu, acest program foloseşte **setflags()** pentru a activa indicatorii **showbase** şi **showpos**:

```
#include <iostream.h>
#include <iomanip.h>
main()

{
    cout << setiosflags(ios::showpos);
    cout << setiosflags(ios::showbase);
    cout <, 123 << " " << hex << 123;

    return 0;
}
```

Manipulatorul **setiosflags()** efectuează aceeaşi acţiune ca şi funcţia membru **setf()**.

Supraîncărcarea operatorilor << şi >>

După cum ştiţi, operatorii << şi >> sunt supraîncărcaţi în C++ pentru a efectua operaţii de I/O pentru tipurile încorporate în acesta. Puteţi şi dvs. să supraîncărcaţi aceşti operatori astfel încât ei să efectueze operaţii de I/O asupra tipurilor pe care le creaţi.

În limbajul C++, operatorul de ieşire << este numit *operator de inserţie* deoarece el introduce caractere într-un stream. Similar, operatorul >> este numit *operator de extragere* deoarece el extrage caractere din acesta. Funcţiile operator care supraîncarcă inserţia şi extragerea sunt numite, în general, *de inserţie* şi, respectiv, *de extragere*.

Crearea propriilor dvs. funcţii de inserţie

Este foarte simplu să creaţi un operator de inserţie pentru o clasă pe care o construiţi. Toate funcţiile de inserţie au această formă generală:

ostream &operator<<(ostream &*stream, tip_clasa obiect*)
{
    // corpul funcţiei de inserţie
    return *stream*;
}

Reţineţi că funcţia returnează o referinţă spre un stream de tipul **ostream**. (Amintiţi-vă că **ostream** este o clasă derivată din **ios** care permite ieşirea.) Primul

parametru al funcției este o referință la streamul de ieșire. Al doilea parametru este obiectul care este inserat. (Al doilea parametru poate fi, și el, o referință spre obiectul care este inserat.) Ultimul lucru pe care trebuie să-l facă funcția înainte de a se încheia este să returneze un *stream*. Aceasta permite ca funcția de inserție să fie folosită într-un lanț de inserții.

În interiorul unei funcții de inserție puteți să introduceți orice fel de proceduri și de operații pe care le doriți. Aceasta înseamnă că este la latitudinea dvs. ce va face exact o asemenea funcție. Totuși, pentru ca ea să se încadreze într-o practică de programare corectă, ar trebui să limitați operațiile efectuate de o funcție de inserție la introducerea de informații într-un stream. De exemplu, probabil că nu este o idee prea bună să aveți o funcție care să calculeze pi cu 30 de zecimale ca efect secundar al unei operații de inserție.

Pentru a vedea un exemplu, să creăm o funcție de inserție pentru obiecte de tipul **agendatelefon**:

```
class agendatelefon {
public:
    char nume[80];
    int codzona;
    int prefix;
    int numar;
    agendatelefon(char *n, int a, int p, int nm)
    {
        strcpy(nume, n);
        codzona = a;
        prefix = p;
        numar = nm;
    }
};
```

Această clasă memorează numele persoanei și numărul de telefon. Iată o cale de a crea o funcție de inserție pentru obiecte de tipul **agendatelefon**:

```
// Afiseaza numele si numarul de telefon
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream <, o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << "-" << o.numar << "\n";

    return stream; // trebuie se returneze stream
}
```


Iată un scurt program care ilustrează funcția de inserție **agendatelefon**:

```
#include <iostream.h>
#include <stream.h>

class agendatelefon {
public:
    char nume[80];
    int codzona;
    int prefix;
    int numar;
    agendatelefon(char *n, int a, int p, int nm)
    {
        strcpy(nume, n);
        codzona = a;
        prefix = p;
        numar = nm;
    }
};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream << o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << " -" << o.numar << "\n";

    return stream; // trebuie sa returneze stream
}

main()
{
    agendatelefon a("Ted", 111, 555, 1234);
    agendatelefon b("Alice", 312, 555, 5768);
    agendatelefon c("Tom", 212, 555, 9991);

    cout << a << b << c;

    return 0;
}
```

Programul are această ieșire:

```
Ted (111) 555-1234
Alice (312) 555-5768
Tom (212) 555-9991
```

Rețineți că, în programul precedent, funcția de inserție pentru **agendatelefon** nu este un membru al clasei **agendatelefon**. Chiar dacă acest lucru pare ciudat la prima vedere, motivul este ușor de înțeles. Când o funcție operator de orice fel este un membru al unei clase, operandul din stânga (transmis implicit prin **this**) este obiectul care generează apelarea acesteia. Mai mult, acest obiect este un *obiect din clasa* în care este membru funcția operator. Nu se poate schimba acest lucru. Dacă o funcție operator supraîncărcată este un membru al unei clase, termenul din stânga trebuie să fie un obiect din acea clasă. Totuși, când supraîncărcați inserțiile, operandul din stânga este un *stream* iar cel din dreapta este un obiect din acea clasă. De aceea, funcțiile de inserție supraîncărcate nu pot fi membri ai clasei pentru care sunt supraîncărcate. De asemenea, singurul motiv pentru care variabilele **nume**, **codzona**, **prefix** și **numar** sunt publice în programul precedent este ca ele să poată fi utilizate de o funcție de inserție care nu este membru al clasei lor.

Faptul că funcțiile de inserție nu pot fi membri ai clasei pentru care sunt definite pare a fi un handicap serios în C++. Dacă aceste funcții supraîncărcate nu sunt membre, cum pot să aibă acces la elementele particulare ale clasei? În programul anterior toți membrii au fost declarați publici. Totuși, încapsularea este o componentă esențială a programării orientate pe obiecte. Cerința ca toate datele care vor fi obținute folosind o funcție de inserție intră în conflict cu acest principiu. Din fericire, există o soluție a acestei dileme: declarați funcția de inserție un **friend** al clasei. Astfel se respectă cerința ca primul argument al funcției de inserție supraîncărcate să fie un stream și se garantează, în continuare, accesul ei la secțiunile particulare ale clasei pentru care este supraîncărcată. Iată același program modificat pentru a avea funcția de inserție ca funcție **friend**:

```
#include <iostream.h>
#include <stream.h>

class agendatelefon {
    // acum particulari
    char nume[80];
    int codzona;
    int prefix;
    int numar;
public:
    agendatelefon(char *n, int a, int p, int nm)
    {
        strcpy(nume, n);
        codzona = a;
        prefix = p;
        numar = nm;
    }
    friend ostream &operator<<(ostream &stream,
                               agendatelefon o);
};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream << o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << "-" << o.numar << "\n";
    return stream; // trebuie sa returneze stream
}

main()
{
    agendatelefon a("Ted", 111, 555, 1234);
    agendatelefon b("Alice", 312, 555, 5768);
    agendatelefon c("Tom", 212, 555, 9991);

    cout << a << b << c;

    return 0;
}
```


Când definiți corpul unei funcții de inserție, amintiți-vă să îl faceți cât mai general posibil. Astfel, cel prezentat în exemplul anterior poate fi folosit cu orice stream deoarece își orientează ieșirea către **stream**, care este streamul care apelează acea funcție. Deși nu este, practic, greșit să scrieți linia

```
stream << o.nume < " ";
```

astfel:

```
cout < o.nume << " ";
```

ea va avea ca efect codificarea hard a streamului **cout** stream de ieșire. Versiunea inițială va lucra cu orice stream, inclusiv cu acelea legate de fișierele de pe disc. Deși, în unele situații, mai ales atunci când sunt implicate echipamente de ieșire speciale,va fi mai bine să codificați hardstreamul de ieșire, în majoritatea ocaziilor nu va fi nevoie. În general, cu cât vor fi mai flexibile funcțiile de inserție, cu atât vor fi mai valoroase.

*NOTĂ: Funcţia de inserţie pentru clasa **agendatelefon** lucrează bine cu excepţia cazurilor în care valoarea din **numar** este ceva de genul 0034, situaţie în care zerourile din faţă nu vor fi afişate. Pentru a rezolva aceasta, puteţi ori să convertiţi **numar** într-un şir, ori să stabiliţi caracterul de umplere ca fiind zero şi să folosiţi funcţia de format **width()** pentru a genera zerourile din faţă. Soluţia este lăsată ca exerciţiu, pentru cititor.*

Înainte de a trece la extractori, să urmărim încă un exemplu de funcţie de inserţie. O astfel de funcţie nu trebuie să fie limitată doar la manevrarea textelor. Ea poate fi folosită pentru a obţine date în orice formă care are sens. De exemplu, o funcţie de inserţie pentru o clasă care face parte din sistemul CAD poate emite instrucţiuni pentru plotter. Alta poate să genereze imagini grafice. O funcţie de inserţie pentru programele în Windows poate să afişeze casete de dialog. Pentru a „simţi" obţinerea altor obiecte, altele decât cele de tip text, să examinăm următorul program, care desenează casete pe ecran. (Deoarece nici C şi nici C++ nu definesc funcţii grafice, programul foloseşte caractere pentru a desena, dar sunteţi liberi să înlocuiţi cu elemente grafice, dacă sistemul vostru le admite.)

```
#include <iostream.h>
class caseta {
    int x, y;
public:
    caseta(int i, int j) {x=i; y=j;}
    friend ostream &operator<<(ostream &stream, caseta o);
};

// Afiseaza o caseta.
ostream &operator<<(ostream &stream, caseta o)
{
    register int i, j;

    for(i=0; i<o.x; i++)
          stream << "*";

    stream << "\n";

    for(j=1; j<o.y-1; j++) {
          for(i=0; i<o.x; i++)
                if(i==0 || i==o.x-1) stream << "*";
                else stream << " ";
          stream << "\n";
    }
    for(i=0; i<o.x; i++)
```



```
          stream << "*";
    stream << "\n";

    return stream;
}

main()
{

    caseta a(14, 6), b(30, 7), c(40, 5);
    cout << "Iata cateva casete:\n";
    cout << a << b << c <<;

    return 0;
}
```

Programul afişează următoarele:

```
Iata cateva casete:
**************
*            *
*            *
*            *
*            *
**************
******************************
*                            *
*                            *
*                            *
*                            *
*                            *
******************************
****************************************
*                                      *
*                                      *
*                                      *
****************************************
```

Crearea propriilor extractori

Extractorii sunt complemente ale funcţiilor de inserţie. Forma generală a funcţiilor de extracţie este:

istream &operator>>(istream &*stream*, *tip_clasa* &*obiect*)
{
    // corpul extractorului
    return *stream*;
}

Extractorii returnează o referință către un stream de tipul **istream**, care este un stream de intrare. Primul parametru trebuie să fie, de asemenea, o referință la un stream de tipul **istream**. Rețineți că al doilea parametru trebuie să se refere la un obiect de clasa pentru care este supraîncărcat extractorul. Aceasta permite ca obiectul să fie modificat de operația de intrare (extragere).

Continuând cu clasa **agendatelefon**, iată o cale de a scrie o funcție de extracție:

```
istream &operator>>(istream, agendatelefon &o)
{
    cout << "Introduceti nume: ";
    stream >> o.nume;
    cout << "Introduceti codzona: ";
    stream >> o.codzona;
    cout << "Introduceti prefix: ";
    stream >> o.prefix;
    cout << "Introduceti numar: ";
    stream >> o.numar;
    cout << "\n";

    return stream;
}
```

Rețineți că, deși aceasta este o funcție de intrare, ea efectuează și ieșire prin solicitarea către utilizator. Chiar dacă scopul principal al extractorului este intrarea, el poate efectua orice operație necesară pentru a-l atinge. Totuși, ca și pentru funcțiile de inserție, este mai bine să mențineți acțiunile efectuate de către extractor în sfera operațiilor de intrare. Dacă nu o veți face, riscați să pierdeți mult din structurare și din claritate.

Iată un program care ilustrează extractorul **agendatelefon**:

```
#include <iostream.h>
#include <string.h>

class agendatelefon {
    char nume[80];
    int codzona;
    int prefix;
    int numar;
public:
    agendatelefon() { };
    agendatelefon(char *n, int a, int p, int nm)
    {
        strcpy(nume, n);
        codzona = a;
        prefix = p;
        numar = nm;
    }
    friend ostream &operator<<(ostream &stream,
                               agendatelefon o);
    friend istream &operator>>(istream &stream,
                               agendatelefon &o);
};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream << o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << "-" << o.numar << "\n";

    return stream; // trebuie sa returneze stream
}

// Introduce numele si numarul de telefon.
istream &operator >>(istream, agendatelefon &o)
{
    cout << "Introduceti nume: ";
    stream >> o.nume;
    cout << "Introduceti codzona: ";
    stream >> o.codzona;
    cout << "Introduceti prefix: ";
    stream >> o.prefix;
    cout << "Introduceti numar: ";
    stream >> o.numar;
    cout << "\n";

    return stream;
}
```

```
main()
{
    agendatelefon a;

    cin >> a;

    cout << a;

    return 0;
}
```

Crearea propriilor dvs. funcţii de manipulare

În afară de supraîncărcarea operatorilor de inserţie şi de extragere, puteţi să vă adaptaţi sistemul de I/O din C++ creând propriile dvs. funcţii de manipulare. Acestea sunt importante din două motive principale. Primul, puteţi să sintetizaţi o secvenţă de mai multe operaţii de I/O individuale într-o singură funcţie de manipulare. De exemplu, apar des situaţii în care într-un program se întâlneşte frecvent aceeaşi secvenţă de operaţii de I/O. În aceste cazuri, puteţi folosi un manipulator construit de dvs. pentru a efectua aceste acţiuni, simplificând astfel codul sursă şi prevenind erorile accidentale. Un manipulator propriu poate, de asemenea, să fie important când trebuie să efectuaţi operaţii de I/O pentru un echipament care nu este standardizat. De exemplu, puteţi să folosiţi un manipulator pentru a transmite codurile de control unui tip special de imprimantă sau unui sistem de recunoaştere optic.

Manipulatorii adaptaţi sunt caracteristici ale lui C++ care admit OOP, dar care pot totodată să trateze programe care nu sunt orientate pe obiecte. După cum veţi vedea, ei vă pot ajuta să faceţi orice program ce foloseşte mult operaţii de I/O mai clar şi mai eficient.

După cum ştiţi, există două tipuri de manipulatori: cei care operează cu streamuri de intrare şi cei care operează cu streamuri de ieşire. Dar, în afară de aceste două mari categorii, mai este şi o a doua împărţire: cei care preiau argumente şi cei care nu o fac. Există câteva diferenţe semnificative între modul în care sunt creaţi manipulatorii fără parametri şi cei parametrizaţi. Acest paragraf discută cum se creează ambele tipuri, începând cu manipulatorii fără parametri.

Crearea manipulatorilor fără parametri

Toate funcţiile de manipulare a ieşirilor fără parametri au această formă:


```
ostream &nume-manip(ostream &stream)
{
    // aici se afla codul dvs.
    return stream;
}
```

nume-manip este aici numele manipulatorului. Reţineţi că este returnată o referinţă la un stream de tip **ostream**. Acest lucru este necesar dacă manipulatorul este folosit ca parte a unei expresii mai mari de I/O. Este important să notaţi că, deşi manipulatorul are drept unic argument o referinţă la un stream asupra căruia operează, când manipulatorul este introdus într-o operaţie de ieşire nu este folosit nici un argument.

Ca un prim exemplu simplu, următorul program creează un manipulator numit **punehex()**, care activează indicatorul **showbase** şi obţine ieşirea în hexazecimal.

```
#include <iostream.h>
#include <iomanip.h>

// Un simplu manipulator de iesire.
ostream &punehex(ostream &stream)
{
    stream.setf(ios::showbase);

    stream.setf(ios::hex);
    return stream;
}

main()
{
    cout << 256 << " " << punehex << 256;
    return 0;
}
```

Acest program afişează **256 0x100**. După cum puteţi vedea, **punehex** este folosit ca parte a unei expresii de I/O în acelaşi fel ca şi oricare dintre manipulatorii încorporaţi.

Pentru a fi folositori, manipulatorii adaptaţi nu trebuie neapărat să fie complecşi. De exemplu, manipulatorii simpli **ss()** şi **sd()** afişează o săgeată la stânga şi respectiv una la dreapta, pentru a scoate ceva în evidenţă, aşa cum se prezintă aici:

```
#include <iostream.h>
#include <iomanip.h>
```

```
// Sageata la dreapta
ostream &sd(ostream &stream)
{
    stream << "-------> ";
    return stream;
}

// Sageata la stanga
ostream &ss(ostream &stream)
{
    stream << " <-------";
    return stream;
}

main()
{
    cout << "Bilant mare" << sd << 1233.33 << "\n";
    cout << "Fara acoperire " << sd << 567.66 << ss

    return 0;
}
```

Acest program afişează:

```
Bilant mare -------> 1233.23
Fara acoperire -------> 567.66 <-------
```

Dacă îi utilizaţi frecvent, aceşti manipulatori vă salvează de la tastări plicticoase.

Utilizarea unor funcţii de manipulare pentru ieşire este folositoare în special pentru a transmite coduri speciale către un echipament. De exemplu, o imprimantă poate fi capabilă să accepte diferite coduri care modifică mărimea tipului sau fontul, sau care plasează capul de scriere într-o anumită poziţie. Dacă aceste modificări trebuie făcute frecvent, atunci ele se pretează foarte bine prelucrării cu un manipulator.

Toate funcţiile de manipulare pentru intrare fără parametri au următoarea formă:

istream &*nume-manip*(istream &*stream*)
{
    // aici este codul dvs.
    return *stream*;
}

Un manipulator de intrare primeşte o referinţă spre streamul pentru care este apelat. Acest stream trebuie să fie returnat de către manipulator.

Următorul program creează manipulatorul de intrare **daparola()**, care sună clopoţelul şi apoi solicită o parolă.

```
#include <iostream.h>
#include <string.h>

// Un simplu manipulator de intrare.
istream &daparola(istream &stream)
{
    cout << '\a'; // suna clopotel
    cout << "Introduceti parola: ";

    return stream;
}

main()
{
    char par[80];

    do {
        cin >> daparola >> par;
    } while (strcmp(par, "parola"));

    cout << "Parola corecta\n";

    return 0;
}
```

Este esenţial ca manipulatorul să returneze un **stream**. Dacă nu o face, nu poate fi folosit într-o secvenţă de operaţii de intrare sau de ieşire.

Crearea manipulatorilor parametrizaţi

Crearea unei funcţii de manipulare care preia un argument este mai puţin intuitivă decât crearea uneia fără parametri. Un motiv pentru aceasta este că manipulatorii parametrizaţi folosesc clasele generice. Acestea sunt create folosindu-se cuvântul cheie **template**. (Şabloanele **(template)** şi clasele generice sunt discutate în **Capitolul 20**.) Dacă nu înţelegeţi cum operează clasele generice, nu veţi fi capabili să înţelegeţi pe deplin crearea manipulatorilor parametrizaţi. Un alt motiv pentru care manipulatorii parametrizaţi sunt mai complecşi este acela că metoda exactă pe care o folosiţi pentru a crea unul diferă de la compilator la compilator.

Deci, ce va funcționa pentru un compilator nu va lucra, neapărat, și la altul. Deși comitetul ANSI C++ va finaliza acest subiect, nu toate compilatoarele introduc manipulatorii parametrizați în corespondență cu standardul. Datorită acestor probleme, dacă veți crea proprii dvs. manipulatori parametrizați, va trebui să consultați manualul compilatorului pentru detalii specifice.

NOTĂ: Exemplele de manipulatori parametrizați prezentate în acest paragraf corespund propunerii de standard ANSI C++ și vor funcționa pentru majoritatea compilatoarelor de C++ moderne.

Pentru a crea un manipulator parametrizat trebuie să includeți în fișierul dvs. IOMANIP.H. În acesta sunt definite trei clase generice: **omanip**, **imanip** și **smanip**. **omanip** este folosit pentru a crea manipulatori de ieșire care preiau un argument. **imanip** este folosită pentru a crea manipulatori de intrare parametrizați ce pot fi folosiți în streamuri de intrare și de ieșire, iar **smanip** pentru manipulatori parametrizați ce acționează atât asupra intrărilor cât și a ieșirilor. (Pentru a vedea cum sunt implementate, probabil că va trebui să vedeți definirea acestor clase din versiunea asigurată de compilatorul dvs. pentru IOMANIP.H.)

În general, întotdeauna când doriți să obțineți un manipulator care preia un argument, va trebui să creați două funcții de manipulare supraîncărcate. În una dintre ele este necesar să definiți doi parametri. Primul parametru este o referință către stream, iar al doilea este cel care va fi transmis funcției. A doua versiune a manipulatorului definește doar un singur parametru - cel care este specificat atunci când manipulatorul este folosit într-o expresie de I/O. Această a doua versiune generează o apelare a primei versiuni. Pentru manipulatorii de ieșire parametrizați veți folosi aceste forme generale:

```
ostream &nume-manip(ostream &stream, tip param)
{
        // aici este codul dvs.
      return stream;
}
      // Supraîncărcare
      omanip<tip>nume-manip(tip param) {
            return omanip<tip>(nume-manip, param);
}
```

nume-manip este aici numele manipulatorului iar *tip* specifică tipul parametrului folosit de manipulator. Deoarece **omanip** este o clasă generică, *tip* devine, de asemenea, tipul de date asupra căruia operează obiectul **omanip** specific returnat de către manipulator.

Următorul program creează un manipulator cu parametri numit **compara()**, care indentează un rând cu numărul specificat de spații.


```
#include <iostream.h>
#include <iomanimp.h>

// Indenteaza cu numarul de spatii.
ostream &indent(ostream &stream, int lungime)
{
   register int i;

   for(i=0; i<lungime; i++) cout << " ";
   return stream;
}

omanip<int> indent(int lungime)
{
   return omanip<int>(indent, lungime);
}

main()
{
   cout << indent(10) << "Acesta este un test\n";
   cout << indent(20) << "pentru manipulatorul de
                           indentare.\n";
   cout << indent(5) << "Merge!\n";
   return 0;
}
```

După cum puteți vedea, **indent()** este suprapusă așa cum s-a descris anterior. Când este întâlnită **indent(10)** în expresia de ieșire, se execută a doua versiune pentru **indent()**, transmițându-se valoarea 10 în parametrul **lungime**. Această versiune execută prima versiune, cu valoarea 10 din nou transmisă prin parametrul **lungime**. Procesul se repetă pentru fiecare apelare a funcției **indent()**.

Tipul parametrului pe care doriți să îl aibă manipulatorul este sub controlul dvs. și este determinat de tipul specificat ca al doilea parametru al funcției de manipulare. Același tip este apoi folosit când se creează clasa generică pentru manipulare. De exemplu, următorul program arată versiunea supraîncărcată a manipulatorului. De exemplu, următorul program arată versiunea supraîncărcată a manipulatorului. Se obține apoi cum se transmite o valoare **double** unei funcții de manipulare. Se obține apoi valoarea folosindu-se un format dolari-și-cenți.

```
#include <iostream.h>
#include <iomanip.h>

ostream &dolari(ostream &stream, double cantitate)
{
```

```
    stream.setf(ios::showpoint);
    stream << "$" << setw(10)  << setprecision(2)
           << cantitate;

    return stream;
}

omanimp <double> dolari(double cantitate) {
    return omanimp<double> (dolari, cantitate);
}
main()
{
    cout << dolari(123.123456);
    cout << "\n" << dolari(10.0);
    cout << "\n"; << dolari(1234.23);
    cout << "\n" << dolari(0.0);

    return 0;
}
```

Şi manipulatorii de intrare pot să preia un parametru. Următorul program dezvoltă manipulatorul **daparola()** prezentat mai înainte. Această versiune preia un argument care specifică de câte încercări beneficiază utilizatorul pentru a introduce corect parola.

```
// Acest program foloseste un manipulator pentru a
// introduce o parola
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>

char *parola="ImiplaceC++";
char par[80];

// Introduce o parola
istream &daparola(istream &stream, int incercari)
{
    do {
          cout << "Introduceti parola: ";
          stream >> par;
          if(!strcmp(parola, par)) return stream;
          cout << "\a"; // clopotel
```



```
          incercari--;
    } while(incercari>0);

    cout << "Toate incercarile eronate!\n";
    exit(1) ; // nu s-a introdus parola

    return stream;
}

imanip<int> daparola(int incercari) {
    return imanip<int>(daparola, incercari);
}

main()
{
    // ofera trei incercari pentru introducerea parolei
    cin >> daparola(3);
    cout << "Parola corecta\n";

    return 0;
}
```

Reţineţi că formatul este acelaşi ca şi pentru manipulatorii de ieşire, cu două excepţii: trebuie folosit streamul de intrare **istream** şi se specifică clasa **imanip**.

Pe măsură ce veţi lucra cu C++, veţi vedea că manipulatorii adaptaţi vă pot ajuta în crearea instrucţiunilor de I/O.

O notiţă despre vechea bibliotecă de clase pentru streamuri

Când a fost inventat C++, a fost creată o bibliotecă de clase de I/O mică şi puţin diferită. Această bibliotecă este definită în fişierul STREAM.H. Însă, o dată cu evoluţia limbajului C++, ea a fost înlocuită de biblioteca de I/O descrisă în carte. Majoritatea compilatoarelor de C++ mai admit încă biblioteca veche pentru streamuri din motive de compatibilitate cu programele vechi în C++. Dar, ar trebui ca, atunci când scrieţi programe noi, să folosiţi biblioteca modernă de I/O definită în IOSTREAM.H.

Capitolul 18

I/O cu fişiere în C++

Chiar dacă abordarea I/O din C++ formează un sistem integrat, operaţiile de I/O cu fişiere (mai precis, cele de I/O pe disc) sunt suficient de specializate pentru a fi considerate un caz aparte, cu cerinţele şi particularităţile sale. Aceasta se întâmplă, în parte, deoarece cel mai uzual fişier este cel de pe disc, iar acesta are caracteristici şi facilităţi pe care alte fişiere nu le au. Reţineţi, totuşi, că operaţiile de I/O pentru fişiere sunt un simplu caz special al sistemului de I/O şi că mare parte din materialul discutat în acest capitol se aplică, de asemenea, şi streamurilor conectate la alte tipuri de echipamente.

fstream.h şi clasele de fişiere

Pentru a efectua I/O cu fişiere, trebuie să includeţi în programul dvs. fişierul antet FSTREAM.H. El defineşte mai multe clase, printre care **ifstream**, **ofstream** şi **fstream**. Aceste clase sunt derivate din **istream** şi, respectiv, din **ostream**. Amintiţi-vă că **istream** şi **ostream** sunt derivate din **ios**, astfel încât **ifstream**, **ofstream** şi **fstream** au, de asemenea, acces la toate operaţiile definite de această clasă de bază (prezentată în capitolul precedent).

Deschiderea şi închiderea unui fişier

Un fişier se deschide în C++ legându-l de un stream. Înainte de a putea să deschideţi un fişier, trebuie, pentru început, să aveţi un stream. Există trei tipuri de streamuri: de intrare, de ieşire şi de intrare/ieşire. Pentru a crea un stream de intrare, trebuie să-l declaraţi ca fiind din clasa **ifstream**. Pentru a crea unul de ieşire, trebuie să-l declaraţi ca fiind din clasa **ofstream**. Streamurile care vor efectua atât operaţii de intrare cât şi de ieşire trebuie declarate ca fiind de clasa **fstream**. De exemplu, acest fragment creează un stream de intrare, unul de ieşire şi unul capabil atât de intrare cât şi de ieşire.

```
ifstream intra; // intrare
ofstream iese; // iesire

fstream inies; // intrare si iesire
```

O dată ce aţi creat streamul, o cale de a-l asocia unui fişier este de a folosi funcţia **open()**. Această funcţie este membru al fiecăreia dintre cele trei clase stream. Prototipul său este:

void open(const char **numefisier*, int *mod*, int *acces=filebuf::openprot*);

numefisier este, aici, un nume de fişier, care poate include specificarea căii de acces. Valoarea din *mod* determină modul de deschidere a fişierului. Modul poate avea una (sau mai multe) dintre aceste valori:

ios::app
ios::ate
ios::binary
ios::in
ios::nocreate
ios::noreplace
ios::out
ios::trunc

Puteţi combina două sau mai multe dintre aceste valori unindu-le prin OR logic. Să vedem ce semnificaţie au ele.

Includerea valorii **ios::app** determină ca toate ieşirile către acel fişier să fie adăugate la sfârşit. Această valoare poate fi folosită doar cu fişierele ce acceptă ieşiri. Includerea valorii **ios::ate** determină căutarea sfârşitului fişierului la deschiderea sa. Cu toate acestea, pentru **ios::ate** operaţiile de I/O pot avea loc oriunde în interiorul fişierului.

Implicit, fişierele se deschid în mod text. Valoarea **ios::binary** determină deschiderea unui fişier în mod binar. Când fişierul este deschis în mod text, au loc diverse modificări de caractere, cum ar fi conversia secvenţei început de rând / salt la linie nouă. Însă, când un fişier este deschis în mod binar, nu apar asemenea modificări. Orice fişier poate fi deschis în mod text sau binar, chiar dacă el conţine un text formatat sau date binare brute. Singura diferenţă priveşte transformarea caracterelor.

Valoarea **ios::in** specifică faptul că fişierul acceptă intrări, iar valoarea **ios::out** că acceptă ieşiri. Totuşi, crearea unui stream folosind **ifstream** implică intrări iar crearea unuia cu **ofstream** implică ieşiri, astfel că, în aceste cazuri, nu mai este necesar să introduceţi aceste valori.

Includerea valorii **ios::nocreate** determină ca funcţia **open()** să eşueze dacă fişierul nu există deja. Valoarea **ios::noreplace** determină ca funcţia **open()** să eşueze dacă fişierul există deja.

Valoarea **ios::trunc** determină distrugerea şi reducerea la zero a mărimii conţinutului unui fişier preexistent care are acelaşi nume cu cel specificat.

***NOTĂ:** Standardul ANSI C++ propus specifică faptul că tipul parametrului mod trebuie să fie **openmod**, care este o formă de întreg (în general, un **int**). În mod curent, majoritatea implementărilor indică pur şi simplu că tipul parametrului mod este **int**.*

Valoarea din *acces* determină modul în care se obţine accesul la fişier. Valoarea implicită este **filebuf::openprot**, care specifică un fişier normal. (**filebuf** este o clasă derivată din **streambuf**.) În majoritatea cazurilor veţi permite *accesul* ca mod implicit. Totuşi, va trebui să verificaţi în manualul compilatorului ce alte opţiuni sunt valabile pentru acest parametru în mediul dvs. de operare. De

exemplu, în mediile pentru rețele, parametrul **acces** specifică de obicei opțiunile de departajare a fişierelor.

Următorul fragment deschide un fişier de ieşire normal.

```
ofstream out;

out.open("test", ios::out);
```

Totuşi, veți vedea rareori (sau niciodată) **open()** apelat aşa cum se arată aici, deoarece parametrul *mod* are şi valori implicite. Pentru **ifstream**, valoarea implicită pentru *mod* este **ios::in**, iar pentru **ofstream**, **ios::out**. De aceea, instrucțiunea anterioară va arăta, de obicei, astfel:

```
out.open("test"); // Au valoarea implicita pentru iesire
```

Pentru a deschide un fişier de intrare şi de ieşire, trebuie să specificați pentru *mod* atât valoarea **ios::in** cât şi **ios::out**, cum se arată în următorul exemplu. (În acest caz nu există nici o valoare implicită pentru *mod*.)

```
fstream streamulmeu;
streamulmeu.open("test", ios::in | ios::out);
```

Dacă **open()** eşuează, **streamulmeu** va fi zero. De aceea, înainte de a folosi un fişier, ar trebui să efectuați un test pentru a vă asigura că operația de deschidere a reuşit. Puteți face aceasta folosind o instrucțiune de tipul:

```
if(!streamulmeu) {
    cout << "Nu pot deschide fisierul.\n";
    // trateaza eroarea
}
```

Chiar dacă este pe deplin corect să deschideți un fişier folosind funcția **open()**, de cele mai multe ori nu o veți face, deoarece clasele **ifstream**, **ofstream** şi **fstream** au funcții constructor care deschid automat fişierul. Funcțiile constructor au aceiaşi parametri şi activări implicite ca şi funcția **open()**. De aceea, de obicei veți vedea un fişier deschis aşa cum se arată aici:

```
ifstream streamulmeu("fisierulmeu"); // deschide fisierul
                                     // pentru intrare
```

După cum am mai spus, dacă, din diferite motive, fişierul nu poate fi deschis, valoarea variabilei stream asociate va fi zero. De aceea, indiferent dacă pentru a deschide un fişier folosiți o funcție constructor sau o apelare explicită a funcției **open()**, ar trebui să testați valoarea streamului pentru a vi se confirma faptul că fişierul s-a deschis efectiv.

Pentru a închide un fişier, folosiți funcția membru **close()**. De exemplu, pentru a închide fişierul legat de streamul numit **streamulmeu**, folosiți această instrucțiune:

```
streamulmeu.close();
```

Funcția **close()** nu preia nici un parametru şi nu returnează nici o valoare.

Citirea şi scrierea fişierelor de text

Este foarte uşor să citiți şi să scrieți un fişier de text. Folosiți pur şi simplu operatorii << şi >> în acelaşi fel în care o faceți şi pentru I/O de la consolă, doar că, în loc să folosiți **cin** şi **cout**, îi veți înlocui cu un stream care este legat de un fişier. De exemplu, acest program creează un fişier scurt pentru inventariere, care conține numele fiecărui articol şi prețul său:

```
#include <iostream.h>
#include <fstream.h>

main()
{
    ofstream out("INVENTAR"); // iesire, fisier normal

    if(!out) {
        cout << "Nu pot deschide fisierul INVENTAR.\n";
        return 1;
    }

    out << "Radio " << 39.35 << endl;
    out << "Prajitor " << 19.95 << endl;
    out << "Mixer " << 24.80 << endl;

    out.close();
    return 0;
}
```

Următorul program citeşte fişierul de inventariere creat de programul anterior şi îi afişează conținutul pe ecran.

```
#include <iostream.h>
#include <fstream.h>
```

```
main()
{
    ifstream in("INVENTAR"); // intrare
    if(!in) {
          cout << "Nu pot deschide fisierul INVENTAR.\n";
          return 1;
    }

    char articol[20];
    float pret;
    in >> articol >> pret;
    cout << articol << " " << pret << "\n";
    in >> articol >> pret;
    cout << articol << " " << pret << "\n";
    in >> articol >> pret;
    cout << articol << " " << pret << "\n";

    in.close();
    return 0;
}
```

Într-un fel, citirea şi scrierea fişierelor cu >> şi << este asemănătoare funcţiilor **fprintf()** şi **fscanf()** din C. Toate informaţiile sunt memorate în fişier în acelaşi format în care ar fi fost afişate pe ecran.

Urmează un alt exemplu de I/O pe disc. Acest program citeşte şiruri introduse de la tastatură şi le scrie pe disc. Programul se opreşte când utilizatorul introduce o linie goală. Pentru a folosi programul, specificaţi pe linia de comandă numele fişierului de ieşire.

```
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>

main(int argc, char *argv[])
{
    if(argc!=2) {
          cout << "Utilizare: iesire <numefisier>\n";
          return 1;
    }

    ofstream out(argv[1]); // iesire, fisier normal

    if(!out) {
          cout << "Nu pot deschide fisierul de iesire.\n";
          return 1;
    }

    char sir[80];
    cout << "Scrie siruri pe disc, RETURN pentru a opri
              programul.\n";

    do {
          cout << ": ";
          gets(sir);
          out << sir << endl;
    }     while (*sir);

    out.close();
    return );
}
```


Când citiţi fişiere de tip text folosind operatorul >>, reţineţi că vor apărea anumite modificări de caractere. De exemplu, sunt omise caracterele de spaţii libere. Dacă doriţi să eliminaţi orice astfel de modificare, trebuie să folosiţi funcţiile de I/O binare din C++, discutate în paragraful următor.

Dacă, atunci când introduceţi, se întâlneşte un sfârşit de fişier, streamul legat de acel fişier va fi zero. (Acest lucru va fi ilustrat în următorul paragraf.)

I/O de tip binar

Există două feluri de a scrie şi de a citi datele de tip binar dintr-un fişier. Ambele metode vor fi explicate în continuare.

***REŢINEŢI:** Dacă veţi efectua operaţii binare asupra unui fişier, aveţi grijă să-l deschideţi folosind modul de specificare **ios::binary**. Chiar dacă funcţiile fişierelor de tip binar vor lucra cu fişierele deschise în modul text, vor apărea unele modificări de caractere, care neagă scopul operaţiilor asupra fişierelor de tip binar.*

get() şi put()

O cale prin care puteţi să citiţi şi să scrieţi date binare este utilizarea funcţiilor membre **get()** şi **put()**. Aceste funcţii sunt orientate pe octeţi, ceea ce înseamnă că **get()** va citi un octet de date, iar **put()** va scrie un octet de date. Funcţia **get()** are

multe forme, dar aici este prezentată, împreună cu **put()**, cea mai des folosită versiune:

istream &get(char &*ch*);
ostream &put(char *ch*);

Funcția **get()** citește un singur caracter din streamul asociat și memorează valoarea în *ch*. Ea returnează o referință către stream. Funcția **put()** scrie *ch* în stream și returnează streamului o referință.

Următorul program afișează pe ecran conținutul unui fișier. El folosește funcția **get()**:

```
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
          cout << "Utilizare: PR <numefisier>\n";
          return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
          cout << "Nu pot deschide fisierul. ";
          return 1;
    }
    while(in) { // cand se ajunge la sfarsit de fisier -
                // EOF - in va fi 0
          in.get(ch);
          cout << ch;
    }

    return 0;
}
```

După cum s-a spus în paragraful precedent, când se ajunge la sfârșitul fișierului, streamul asociat fișierului devine zero. De aceea, când **in** ajunge la sfârșit, streamul va fi zero, determinând oprirea buclei **while**.

Există, de fapt, un mod mai compact de a scrie bucla care citește și afișează un fișier, așa cum se prezintă aici:


```
while(in.get(ch))
    cout << ch;
```

Aceasta va lucra deoarece **get()** returnează o referință către streamul **in**, iar acesta va fi zero când se va întâlni sfârșitul fișierului.

Următorul program folosește **put()** pentru a scrie toate caracterele de la zero la 255 într-un fișier numit CHARS. După cum probabil știți, caracterele ASCII ocupă doar aproximativ jumătate din valorile care pot fi păstrate într-un **char**. Celelalte valori sunt, de obicei, numite *set de caractere extins* și includ printre altele caractere din limbi străine și simboluri matematice. (Nu toate sistemele admit setul de caractere extins, dar majoritatea o fac.)

```
#include <iostream.h>
#include <fstream.h>
main()
{
    int i;
    ofstream out("CHARS", ios::out | ios::binary);

    if(!out) {
          cout << "Nu pot deschide fisierul.\n";
          return 1;
    }

    // scrie toate caracterele pe disc
    for(i=0; i<257; i++) out.put((char) i);
    out.close();
    return 0;
}
```

Poate că veți găsi interesantă ideea să verificați conținutul fișierului CHARS pentru a vedea ce caractere extinse are calculatorul dvs.

read() și write()

A doua cale de a citi și a scrie blocuri de date în binar este folosirea funcțiilor din C++ **read()** și **write()**. Prototipurile lor sunt:

istream &read(unsigned char **buf*, int *numar*);
ostream &write(const unsigned char **buf*, int *numar*);

Funcția **read()** citește *numar*-ul de octeți din streamul asociat și îi pune în bufferul indicat de *buf*. Funcția **write()** scrie în streamul asociat *numar*-ul de octeți citiți din bufferul spre care indică *buf*.

NOTĂ: Standardul propus pentru ANSI C++ specifică tipul parametrului numar ca fiind ***streamsize****, care este un* ***typedef*** *pentru un tip de întreg. În mod curent majoritatea compilatoarelor de C++ indică pur şi simplu că numar este un întreg, aşa cum se arată în prototipurile anterioare. În general, propunerea de standard ANSI C++ foloseşte* ***streamsize*** *ca tip al oricărui obiect care specifică numărul de octeţi transferaţi printr-o operaţie de intrare sau de ieşire.*

Următorul program scrie o structură pe disc, apoi o citeşte.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

struct stare {
   char nume[80];
   float bilant;
   unsigned long numar_cont;
};

main()
{
   struct stare ct;

   strcpy(ct.nume, "Ralph Trantor");
   ct.bilant = 1123.23;
   ct.numar_cont = 34235678;

   ofstream outbal("bilant", ios::out | ios::binary);

   if(!outbal) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
   }

   outbal.write((unsigned char *) &ct, sizeof(struct stare));
   outbal.close();

   // acum, citeste ce a scris
   ifstream inbal("bilant", ios::in | ios::binar);

   if(!inbal) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
   }

   inbal.read((unsigned char *) &ct, sizeof(struct stare));

   cout << ct.nume << endl;
   cout << "Cont # " << ct.numar_cont;
   cout.precision(2);
   cout.setf(ios::fixed);
   cout << endl << "Bilant: $" << ct.bilant;

   inbal.close();
   return 0;
}
```


După cum puteţi vedea, este necesară o singură apelare a funcţiilor **read()** şi **write()** pentru a citi sau scrie întreaga structură. Nu este nevoie ca fiecare câmp individual să fie citit sau scris separat. După cum ilustrează acest exemplu, bufferul poate fi orice tip de obiect.

NOTĂ: Când ***read()*** *şi* ***write()*** *operează asupra unui buffer care nu este definit ca o matrice de caractere, în interiorul apelărilor lor, sunt necesari modelatorii de tip. Datorită stricteţii de verificare a tipului din C++ un pointer de un anumit tip nu va fi convertit automat într-unul de alt tip.*

Dacă se ajunge la sfârşitul fişierului înainte de a fi citite *numar* de caractere, atunci **read()** se opreşte iar bufferul conţine atâtea caractere câte a înregistrat. Puteţi şti câte caractere au fost citite folosind o altă funcţie membru, numită **gcount()**, care are următorul prototip:

int gcount();

Ea returnează numărul de caractere citite de ultima operaţie de intrare binară. Următorul program arată un alt exemplu de folosire a funcţiilor **read()** şi **write()** şi ilustrează utilizarea lui **gcount()**.

```
#include <iostream.h>
#include <fstream.h>

main(void)
{
   float fnum[4] = {99.75, -34.4, 1776.0, 200.1};
   int i;
```

```
    ofstream out("numere", ios::out | ios::binary);
    if(!out) {
          cout << "Nu pot deschide fisierul.\n";
          return 1;
    }

    out.write((unsigned char *) &fnum, sizeof fnum);

    out.close();

    for(i=0; i<4; i++) // sterge matricea
          fnum[i] = 0.0;

    ifstream in("numere", ios::in | ios::binary);
    in.read((unsigned char *) &fnum, sizeof fnum);

    // afla cati octeti au fost cititi
    cout << in.gcount() << " octeti cititi\n";

    for(i=0; i<4; i++) // arata valorile citite din fisier
    cout << fnum[i] << " ";

    in.close();

    return 0;
}
```

Acest program scrie pe disc o matrice de valori în virgulă mobilă şi apoi o citeşte. După apelarea funcţiei **read()**, **gcount()** este folosită pentru a determina câţi octeţi au fost citiţi efectiv.

Mai multe funcţii get()

În plus faţă de forma prezentată mai devreme, funcţia **get()** este supraîncărcată în mai multe feluri. Iată prototipurile celor mai des utilizate forme de supraîncărcare:

istream &get(char **buf*, int *numar*, char *delim*='\n');
int get();

Prima formă de supraîncărcare citeşte caractere dintr-o matrice spre care indică *buf*, până când ori se atinge *număr* de caractere, ori se întâlneşte caracterul specificat de *delim*. **get()** va termina cu null matricea spre care indică *buf*. Dacă nu se specifică nici un parametru pentru *delim*, rolul de limitator va fi jucat de

caracterul de linie nouă. Dacă, în streamul de intrare, este întâlnit caracterul de delimitare, el *nu* este extras şi el rămâne în stream până la următoarea operaţie de intrare.

A doua formă de supraîncărcare pentru **get()** returnează din stream caracterul următor. Ea returnează EOF dacă se întâlneşte sfârşitul fişierului. Această formă a funcţiei **get()** este similară funcţiei **get()** din C.

getline()

O altă funcţie membru care efectuează intrări este **getline()**. Prototipul ei este:

istream &getline(char **buf*, int *numar*, char *delim*='\n');

După cum puteţi vedea, această funcţie este practic identică cu versiunea **get()**, **get(buf, num, delim)**. Ea citeşte caractere de la intrare şi le pune într-o matrice spre care indică *buf* până când ori se citesc *numar* de caractere, ori se întâlneşte caracterul specificat de *delim*. Dacă nu este specificat, *delim* este implicit caracterul de linie nouă. Matricea spre care indică *buf* se termină cu null. Diferenţa dintre **get(buf, num, delim)** şi **getline()** este că ultima citeşte şi extrage delimitatorul din streamul de intrare.

Iată un program care ilustrează funcţia **getline()**. El citeşte conţinutul unui fişier text, câte o linie o dată, şi îl afişează pe ecran.

```
// Citeste si afiseaza un fisier text linie cu linie.
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
    if(argc!=2) {
          cout << "Utilizare: Afiseaza<numefisier>\n";
          return 1;
    }
    ifstream in(argv[1]);
    if(!in) {
          cout << "Nu pot deschide fisierul.\n";
          return 1;
    }

    char sir[255];

    while(in) {
          in.getline(sir, 255); // limita implicita '\n'
```

```
        cout << sir << endl;
    }

    in.close();
    return 0;
}
```

Detectarea EOF

Puteţi să detectaţi când s-a ajuns la sfârşitul fişierului folosind funcţia membru **eof()**, care are acest prototip:

int eof();

Ea returnează o valoare nonzero când a fost atins sfârşitul fişierului; altfel, returnează zero.

*NOTĂ: Propunerea pentru standardul ANSI C++ specifică tipul returnat de **eof()** ca fiind **bool**. Totuşi, cele mai uzuale compilatoare de C++ disponibile nu admit tipul de dată **bool**. Din punct de vedere practic nu are importanţă faptul că tipul returnat de **eof** este specificat ca **bool** sau ca **int** deoarece **bool** este ridicat în orice expresie la rang de **int**.*

Următorul program foloseşte **eof()** pentru a afişa conţinutul unui fişier atât în hexazecimal cât şi în cod ASCII.

```
/* Display contents of specified file
   in both ASCII and hex.
*/
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h>

main(int argc, char *argv[])
{
    if(argc(!=2) {
        cout << "Usage: Display <filename>\n";
        return 1;
    }
```



```
    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    register int i, j;
    int count = 0;
    char c[16];

    cout.setf(ios::uppercase);
    while(!in.eof()) {
        for(i=0; i<16 && !in.eof(); i++) {
            in.get(c[i]);
        }
        if(i<16) i--; // get rid of eof

        for(j=0; j<i; j++)
            cout << setw(3) << hex  << (int) c[j];
        for(; j<16; j++) cout << "  ";
        cout << "\t";
        for(j=0; j<i; j++)
            if(isprint(c[j])) cout << c[j];
            else cout << ".";

        cout << endl;

        count++;
        if(count==16) {
            count = 0;
            cout << "Press ENTER to continue: ";
            cin.get();
            cout << endl;
        }
    }

    in.close;
    return 0;
}
```

Când acest program este folosit pentru a se afişa pe sine însuşi, primul ecran arată astfel:

```
2F 2A 20 44 69 73 70 6C 61 79 20 63 6F 6E 74 65  /* Display conte
6E 74 73 20 6F 66 20 73 70 65 63 69 66 69 65 64  nts of specified
20 66 69 6C 65  D  A 20 20 20 69 6E 20 62 6F 74   file..   in bot
68 20 41 53 43 49 49 20 61 6E 64 20 69 6E 20 68  h ASCII and in h
65 78 2E  D  A 2A 2F  D  A 23 69 6E 63 6C 75 64  ex...*/..#includ
65 20 3C 69 6F 73 74 72 65 61 6D 2E 68 3E  D  A  e <iostream.h>..
23 69 6E 63 6C 75 64 65 20 3C 66 73 74 72 65 61  #include <fstrea
6D 2E 68 3E  D  A 23 69 6E 63 6C 75 64 65 20 3C  m.h>..#include <
63 74 79 70 65 2E 68 3E  D  A 23 69 6E 63 6E 75  ctype.h> ..#inclu
64 65 20 3C 69 6F 6D 61 6E 69 70 2E 68 3E  D  A  de <iomanip.h>..
23 69 6E 63 6C 75 64 65 20 3C 73 74 64 69 6F 2E  #include <stdio.
68 3E  D  A  D  A 6D 61 69 6E 28 69 6E 74 20 61  h>....main(int a
72 67 63 2C 20 63 68 61 72 20 2A 61 72 67 76 5B  rgc, char *argv[
5D 29  D  A 7B  D  A 20 20 69 66 28 61 72 67 63  ])..{..  if(argc
21 3D 32 29 20 7B  D  A 20 20 20 20 63 6F 75 74  !=2) {..    cout
20 3C 3C 20 22 55 73 61 67 65 3A 20 44 69 73 70   << "Usage: Disp
Press ENTER to continue:
```

Funcţia ignore()

Puteţi să folosiţi funcţia membru **ignore()** pentru a citi şi a ignora caractere din streamul de intrare. Ea are prototipul acesta:

istream &ignore(int *num*=1, int *delim*=EOF);

Ea citeşte şi elimină caracterele până când sunt ignorate *numar* de caractere (implicit, 1) sau până când se întâlneşte caracterul specificat prin *delim* (implicit, EOF). Dacă se întâlneşte caracterul de delimitare, el nu este extras din streamul de intrare.

Următorul program citeşte un fişier numit TEST. El ignoră caracterele până când se întâlneşte un spaţiu sau până când au fost citite 10 caractere. Apoi afişează restul fişierului.

```
#include <iostream.h>
#include <fstream.h>

main()
{
    ifstream in("test");

    if(!in) {
```



```
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    /* Ignora pana la 10 caractere sau pana este gasit
       primul spatiu. */
    in.ignore(10, ' ');
    char c;
    while(in) {
        in.get(c);
        cout << c;
    }

    in.close();
    return 0;
}
```

peek() şi putback()

Folosind **peek()**, puteţi să obţineţi caracterul următor din streamul de intrare, fără să-l extrageţi de aici. Funcţia are următorul prototip:

int peek();

Ea returnează următorul caracter din stream sau EOF dacă s-a ajuns la sfârşitul fişierului.

Folosind **putback()**, puteţi să returnaţi ultimul caracter citit dintr-un stream înapoi în acelaşi stream. Prototipul ei este:

istream &putback(char *c*);

unde *c* este ultimul caracter citit.

flush()

Când se efectuează ieşirea datelor, ele nu sunt neapărat scrise imediat la echipamentul fizic de care este legat streamul. Informaţiile pot fi memorate într-un buffer intern până când acesta se umple. Doar atunci conţinutul bufferului este scris pe disc. Însă, folosind **flush()**, puteţi forţa ca informaţiile să fie scrise efectiv pe disc înainte ca bufferul să fie plin. Prototipul ei este:

ostream &flush();

Apelările funcţiei **flush()** sunt de dorit atunci când programul rulează într-un mediu neprietenos (de exemplu, în situaţii în care apar frecvent întreruperi de curent).

NOTĂ: Închiderea unui fişier sau terminarea normală a unui program determină, de asemenea, golirea tuturor bufferelor.

Accesul aleator

În sistemul de I/O din C++, accesul aleator se efectuează folosind funcţiile **seekg()** şi **seekp()**. Formele lor cele mai uzuale sunt:

istream &seekg(streamoff *offset*, seek_dir *origine*);
ostream &seekp(streamoff *offset*, seek_dir *origine*);

streamoff este un tip definit în IOSTREAM.H, capabil să conţină cea mai mare valoare validă pe care o poate avea *offset*, iar **seek_dir** este o enumerare care are aceste valori:

ios::beg
ios::cur
ios::end

Sistemul de I/O din C++ operează cu doi pointeri asociaţi unui fişier. Unul este *pointerul get*, care specifică unde va apărea următoarea operaţie de intrare în fişier. Celălalt este *pointerul put*, şi specifică unde va avea loc în fişier următoarea operaţie de ieşire. După fiecare operaţie de intrare sau de ieşire, pointerul corespunzător este avansat secvenţial, automat. Dar folosirea funcţiilor **seekg()** şi **seekp()** vă permite să aveţi acces la fişier într-un mod nesecvenţial.

Funcţia **seek()** deplasează pointerul get al fişierului asociat cu un număr de octeţi egal cu valoarea *offset* faţă de *originea* specificată, care trebuie să aibă una dintre aceste trei valori:

ios::beg	Început de fişier
ios::cur	Locaţie curentă
ios::end	Sfârşit de fişier

Funcţia **seekp()** deplasează pointerul put al fişierului asociat cu un număr de octeţi egal cu valoarea *offset* faţă de *originea* specificată, care trebuie să aibă una dintre valorile de mai sus.

Următorul program ilustrează funcţia **seekp()**. El vă permite să modificaţi un anumit caracter dintr-un fişier. Specificaţi pe linia de comandă un nume de fişier, urmat de numărul octetului din fişier pe care doriţi să-l modificaţi, urmat de noul caracter. Observaţi că fişierul este deschis pentru operaţii de citire/scriere.


```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    if(argc!=4) {
          cout << “Utilizare: MODIFICA <numefisier> <octet>
                    <caracter>\n”;
          return 1;
    }
 fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out){
    cout << “Nu pot deschide fisierul.\n”;
    return 1;
 }

 out.seekp(atoi(argv[2]), ios::beg);

    out.put (*argv[3]);
    out.close();

    return 0;
 }
```

De exemplu, pentru a înlocui prin intermediul acestui program cu **Z** al 12-lea octet din fişierul numit TEST, folosiţi această linie de comandă:

change test 12 Z

Următorul program foloseşte **seekg()**. El afişează conţinutul unui fişier începând cu locaţia pe care o specificaţi pe linia de comandă.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    char ch;
```

```
    if(argc!=3) {
          cout << "Utilizare: ARATA <numefisier> <pozitia
                   initiala>\n";
          return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
          cout << "Nu pot deschide fisierul.";
          return 1;
    }

    in.seekg(atoi(argv[2]), ios::beg);

    while(in.get(ch))
          cout << ch;

    return 0;
}
```

Următorul program foloseşte atât **seekp()** cât şi **seekg()** pentru a inversa primele <numar> de caractere din fişier.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    if(argc!=3) {
          cout << "Utilizare: Invers <numefisier> <numar>\n";
          return 1;
    }

    fstream inout(argv[1], ios::in | ios::out | ios::binary);
    if(!inout) {
          cout << "Nu pot deschide fisierul.\n";
          return 1;
    }

    long e, i, j;
    char c1, c2;
    e = atol(argv[2]);

    for(i=0, j=e; i<j; i++, j--) {
          inout.seekg(i, ios::beg);
          inout.get(c1);
          inout.seekg(j, ios::beg);
          inout.get(c2);

          inout.seekp(i, ios::beg);
          inout.put(c2);
          inout.seekp(j, ios::beg);
          inout.put(c1);
    }

    inout.close();
    return 0;
}
```

Pentru a folosi programul, specificaţi numele fişierului în care doriţi să faceţi inversările, urmat de numărul de caractere de inversat. De exemplu, pentru a inversa primele zece caractere din fişierul numit TEST, folosiţi această linie de comandă:

reverse test 10

Dacă fişierul conţine

Acesta e un test;

după executarea programului el va conţine următoarele:

u e atsecAn test.

Obţinerea poziţiei curente dintr-un fişier

Folosind funcţiile următoare, puteţi determina poziţia curentă a fiecărui pointer al fişierului:

```
streampos tellg();
streampos tellp();
```

streampos este, aici, un tip definit în IOSTREAM.H care este capabil să păstreze cea mai mare valoare pe care o poate returna oricare dintre cele două funcţii.

Puteţi folosi valorile returnate de **tellg()** şi de **tellp()** ca argumente pentru următoarele forme de **seekg()** şi, respectiv, **seekp()**:

istream &seekg(streampos *poz*);
ostream &seekp(streampos *poz*);

Aceste funcţii vă permit să salvaţi poziţia curentă a fişierului, să efectuaţi alte operaţii specifice fişierelor şi apoi să reveniţi în locaţia salvată anterior.

Starea de I/O

Sistemul de I/O din C++ întreţine informaţii de stare relativ la rezultatul fiecărei operaţii de I/O. Starea curentă a sistemului de I/O este păstrată într-un întreg în care sunt codificaţi următorii indicatori:

Nume	Semnificaţie
eofbit	1, când se întâlneşte sfârşitul fişierului 0, altfel
failbit	1, când apare o eroare de I/O(posibil nu fatală) 0, altfel
badbit	1, când apare o eroare fatală 0, altfel

Aceşti indicatori sunt enumeraţi în cadrul clasei **ios**. De asemenea, în **ios** mai este definit şi **goodbit** (fără eroare), care are valoarea 0.

Există două căi prin care puteţi obţine informaţii de stare pentru I/O. Mai întâi, puteţi apela funcţia membru **rdstate()**. Ea are prototipul acesta:

int rdstate();

Ea returnează starea curentă a indicaţiilor de eroare codificaţi într-un întreg. După cum probabil bănuiţi urmărind lista precedentă de indicatori, **rdstate()** returnează zero când nu apare nici o eroare. Altfel, se activează anumiţi biţi de eroare.

*NOTĂ: Standardul ANSI C++ propus specifică tipul returnat de **rdstate()** ca fiind **iostate**, care este un **typdef** pentru o anumită formă de întreg. Uzual, majoritatea compilatoarelor de C++ indică tipul returnat de **rdstate** ca fiind **int**.*

Următorul program ilustrează **rdstate()**. El afişează conţinutul unui fişier text. Dacă apare o eroare, programul o anunţă, folosind **verifstare()**.


```
#include <iostream.h>
#include <fstream.h>

void verifstare(ifstream &in);

main(int argc, char *argv[])
{
    if(argc!=2) {
          cout << "Utilizare: Afiseaza <numefisier>\n";
          return 1;
    }

    ifstream in(argv[1]);
    if(!in) {
          cout << "Nu pot deschide fisierul.\n";
          return 1;
    }
    char c;
    while(in.get(c)) {
          cout << c;
          verifstare(in);
    }

    verifstare(in); // verifica starea finala

    in.close();
    return 0;
}

void verifstare(ifstream &in)
{
    int i;

    i = in.rdstate();
    if(i & ios::eofbit)
          cout << "S-a intalnit EOF\n";
    else if(i & ios::failbit)
          cout << "Eroare I/O nefatala\n";
    else if(i & ios::badbit)
          cout << "Eroare fatala\n";
}
```

Acest program va semnala întotdeauna o singură „eroare". După terminarea buclei **while**, apelarea finală pentru **verifstare()** semnalează, după cum ne

aşteptam, că a fost întâlnit EOF. Poate că veţi găsi utilă funcţie **verifstare()** în programele pe care le scrieţi.

Cealaltă cale prin care puteţi verifica dacă a apărut o eroare este folosirea uneia sau mai multora dintre aceste funcţii:

```
int bad();
int eof();
int fail();
int good();
```

Funcţia **eof()** a fost discutată mai devreme. Funcţia **bad()** returnează adevărat dacă este activat **badbit**. Funcţia **fail()** returnează adevărat dacă **failbit** este activat. Funcţia **good()** returnează adevărat dacă nu există erori. În caz contrar ele returnează fals.

*NOTĂ: Standardul ANSI C++ propus specifică tipul returnat de **bad()**, **eof()**, **fail()** şi **good()** ca **bool**. Dar compilatoarele uzuale specifică tipul returnat de funcţii ca fiind **int**. Din punct de vedere practic, diferenţa este nesemnificativă deoarece, într-o expresie, **bool** este convertit automat în **int**.*

O dată ce a apărut o eroare, ea trebuie să fie îndepărtată înainte ca programul să continue. Pentru a face aceasta, folosiţi funcţia **clear()**, care are următorul prototip:

void clear(int *indicatori*=0);

Dacă *indicatori* este 0 (aşa cum este implicit), toţi indicatorii pentru erori sunt şterşi (iniţializaţi cu zero). Altfel, stabiliţi ca *indicatori* valorile pe care doriţi să le ştergeţi sau indicatorii.

I/O şi fişiere adaptate

În **Capitolul 17** aţi învăţat cum să supraîncărcaţi operatorii de inserţie şi de extracţie relativ la clasele dvs. proprii. În acel capitol au fost efectuate doar I/O pentru consolă. Totuşi, deoarece toate streamurile C++ sunt identice, puteţi folosi aceeaşi funcţie pentru inserţie supraîncărcată pentru ieşiri pe ecran sau într-un fişier, fără nici o modificare. Ca ilustrare, acest program modifică exemplul agendei de telefon din **Capitolul 17**, astfel încât să memoreze o listă pe disc. Programul este foarte simplu: el vă permite să adăugaţi nume pe listă sau să afişaţi lista pe ecran. Totuşi, ca un exerciţiu, poate că veţi găsi că este interesant să dezvoltaţi programul astfel încât să găsească un anumit număr şi să şteargă numere nedorite.


```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

class agendatelefon {
    char nume[80];
    char codzona[4];
    char prefix[4];
    char numar[5];
public:
    agendatelefon() { };
    agendatelefon(char *n, int *a, int *p, int *nm)
    {
            strcpy(nume, n);
            strcpy(codzona, a);
            strcpy(prefix, p);
            strcpy(numar, nm);
    }
    friend ostream &operator<<(ostream &stream,
                                     agendatelefon o);
    friend istream &operator>>(istream &stream,
                                     agendatelefon &o);
};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream << o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << "-";
    stream <<o.num << "\n";
    return stream; // trebuie sa returneze stream
}
// Introduce nume si numere de telefon.
istream &operator>>(istream &stream, agenda telefon &o)
{
    cout << "Introduceti nume: ";
    stream >> o.nume;
    cout << "Introduceti codzona: ";
    stream >> o.codzona;
    cout << "Introduceti prefix: ";
    stream >> o.prefix;
    cout << "Introduceti numar: ";
```

```
    stream >> o.numar;
    cout << "\n";
    return stream;
}

main()
{
    agendatelefon a;
    char c;

    fstream at("telefon", ios::in | ios::out | ios::app);
    if(!at) {
        cout << "Nu pot deschide fisierul agenda de
                telefon.\n";
        return 1;
    }

    for(;;) {
        do {
            cout << "1. Introduceti numerele\n";
            cout << "2. Afiseaza numerele\n";
            cout << "3. Parasiti programul\n";
            cout << "\nIntroduceti o optiune: ";
            cin >> c;
        }   while(c<'1' || c>'3');
        switch(c) {
            case '1':
                cin >> a;
                cout << "Intrarea este: ";
                cout << a; // afiseaza pe ecran
                at << a; // scrie pe disc
                break;
            case '2';
                char ch;
                at.seekg(0, ios::beg);
                while(!at.eof()) {
                    at.get(ch);
                    cout << ch;
                }
                at.clear(); // reset eof
                cout << endl;
                break;
            case '3':
```



```
                at.close();
                return 0;
        }
    }
}
```

Reţineţi că operatorul suprapus << poate fi folosit pentru a scrie într-un fişier de pe disc sau pe ecran, fără nici o schimbare. Aceasta este una dintre cele mai importante şi folositoare facilităţi ale abordării de I/O din C++.

Capitolul 19

I/O bazate pe matrice

În plus față de I/O pentru consolă și fișier, sistemul de I/O din C++ axat pe stream permite operații de I/O *bazate pe matrice*. Operațiile de I/O *bazate pe matrice* folosesc memoria RAM ca dispozitiv de intrare, de ieșire sau și una și alta. Ele sunt efectuate prin streamuri C++ normale. De fapt, toate informațiile prezentate în cele două capitole precedente sunt aplicabile și pentru operațiile de I/O bazate pe matrice. Ceea ce diferă este că dispozitivul care se leagă de stream este memoria.

O parte din literatura de C++ numește operațiile de I/O bazate pe matrice *I/O cu RAM*. De asemenea, deoarece streamurile sunt, ca și toate streamurile de C++, capabile să manevreze informații formatate, operațiile de I/O bazate pe matrice sunt numite și *formatare în RAM*. (Uneori se mai folosește și termenul arhaic *formatare incore*. Totuși, deoarece memoria „core" este de domeniul trecutului, această carte va folosi termenul *formatare în RAM* și *bazată pe matrice*.)

Operațiile de I/O din C++ bazate pe matrice au același efect ca funcțiile **sprintf()** și **sscanf()** din C. Ambele metode folosesc memoria ca pe un dispozitiv de intrare și de ieșire.

Pentru a folosi în programele dvs. operații de I/O bazate pe matrice, trebuie să includeți STRSTREA.H.

Clasele bazate pe matrice

Clasele de I/O bazate pe matrice sunt **istrstream**, **ostrstream** și **strstream**. Aceste clase sunt folosite pentru a crea streamuri de intrare, de ieșire și, respectiv, de intrare/ieșire. Toate aceste clase au **strstreambuf** drept una dintre clasele de bază. Acesta definește foarte multe detalii de lucru la nivel scăzut, care sunt folosite și de clasele derivate. În plus față de **strstreambuf**, clasa **istrstream** are, de asemenea, **istream** ca bază. Clasa **ostrstream** este derivată și din **ostream**, iar clasa **strstream** conține, de asemenea, clasele **iostream**. De aceea, toate clasele bazate pe matrice au acces la aceleași funcții membre la care au acces și clasele „normale" de I/O.

Crearea unui stream de ieșire bazat pe matrice

Pentru a lega un stream la o matrice, folosiți acest constructor **ostrstream**:

ostrstream ostr(char **buffer*, int *marime*, int *mod=ios::out*)

buffer este, aici, un pointer spre o matrice care va fi folosită pentru a păstra caracterele scrise în stream. Mărimea matricei este transmisă prin parametrul *size*. Streamul este deschis implicit pentru ieșire normală, dar puteți să îl uniți prin OR logic de diverse alte opțiuni pentru a crea modul care vă este necesar. (De exemplu, puteți să includeți **ios::app** pentru a determina ca ieșirea să fie scrisă la sfârșitul informațiilor conținute deja în matrice.) În cele mai multe cazuri, *mod* va fi lăsat la valoarea implicită.

O dată ce ați deschis un stream de ieșire bazat pe matrice, toate ieșirile din acest stream sunt trecute într-o matrice. Totuși, nici o ieșire nu va fi scrisă în afara limitelor acesteia. O încercare de a face așa ceva va determina o eroare.

Iată un program simplu, care ilustrează un stream de ieșire bazat pe matrice:

```
#include <strstream.h>
#include <iostream.h>

main()
{
    char sir[80];

    ostrstream iesiri(sir, sizeof(sir));

    iesiri << "Hello ";
    iesiri << 99-14 << hex << " ";
    iesiri.setf(ios::showbase);
    iesiri << 100 << ends;
    cout << sir; // afiseaza sirul la consola

    return 0;
}
```

Acest program afișează **Hello 85 0x64**. Rețineți că **iesiri** este un stream ca oricare altul, cu aceleași facilități. Singura diferență este că dispozitivul la care este legat este memoria. Deoarece **iesiri** este un stream, manipulatorii ca **hex** și **ends** sunt perfect valizi. De asemenea, este permisă utilizarea funcțiilor membre din **ostream**, cum ar fi **setf()**.

Dacă doriți să obțineți matrice terminate cu null, trebuie să scrieți null explicit. În programul precedent manipulatorul **ends** a fost folosit pentru a încheia șirul cu null, dar ați fi putut folosi și **'\0'**.

Dacă nu sunteți foarte sigur de ce se întâmplă efectiv în programul precedent, comparați-l cu următorul program în C. Acesta este funcțional echivalent versiunii în C++, doar că, pentru a construi o matrice de ieșire, folosește **sprintf()**.

```
#include <stdio.h>

main()
{
    char sir[80];

    sprintf(sir, "Hello %d %#x", 99-14, 100);
    printf(sir);
```

```
    return 0;
}
```

Puteţi să determinaţi câte caractere sunt în matricea de ieşire apelând funcţia membru **pcount()**. Ea are următorul prototip:

int pcount();

Numărul returnat de **pcount()** include şi null de încheiere, dacă acesta există.
Următorul program ilustrează **pcount()**. El spune că în **iesiri** sunt 17 caractere - 16 caractere plus caracterul null de încheiere.

```
#include <strstrea.h>
#include <iostream.h>

main()
{
    char sir[80];

    ostrstream iesiri(sir, sizeof(sir));

    iesiri << "Hello ";
    iesiri << 34 << " " << 1234.23;
    iesiri << ends; // null de terminare

    cout << iesiri; // afiseaza numarul de caractere
                    // din iesiri

    cout << " " << sir;

    return 0;
}
```

Utilizarea unei matrice ca intrare

Pentru a lega un stream de intrare de o matrice, folosiţi acest constructor **istrstream**;

istrstream istr(char **buffer*);

buffer este, aici, un pointer spre o matrice care va fi folosită ca sursă de caractere de fiecare dată când se efectuează o intrare în stream. Conţinutul matricei spre

care indică *buffer* trebuie să fie încheiat cu null. Însă, acesta nu este citit niciodată.
Iată un exemplu care foloseşte ca intrare un şir:

```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char s[] = "10 Hello 0x88 12.23 gata";

    istrstream ins(s);

    int i;
    char sir[80];
    float f;

    // citeste: 10 Hello
    ins >> i;
    ins >> sir;

    cout << i << " " << sir << endl;

    // citeste 0x88 12.23 gata
    ins >> i;
    ins >> f;
    ins >> str
    cout << hex << i << " " << f << " " << sir;

    return 0;
}
```

Dacă doriţi ca doar o parte a şirului să fie folosită pentru intrare, utilizaţi această formă de constructor **istrstream**:

istrstream istr(char **buffer*, int *marime*);

Aici vor fi folosite doar primele *marime* elemente ale matricei spre care indică *buffer*. Acest şir nu trebuie să fie terminat cu null deoarece factorul care determină mărimea şirului este valoarea argumentului *marime*.
Streamurile legate de memorie se comportă exact la fel ca şi cele legate la alte echipamente. De exemplu, următorul program ilustrează cum poate fi citit conţinutul unei matrice de tip text. Când se ajunge la sfârşitul matricei (la fel ca pentru fişiere), **ins** va fi zero.

```
/* Acest program arata cum se citeste continutul oricarei
   matrice care contine un text. */
#include <iostream.h>
#include <strstream.h>

main()
{
    char s[] = "10.23 acesta este un text !#?@\n";

    istrstream ins(s);

    char ch;

    /* Aceasta va citi si va afisa continutul
       oricarei matrice tip text. */
    ins.unsetf(ios::skipws); // nu omite spatii
    while (ins) { // 0 cind se ajunge la sfarsitul matricei
        ins >> ch;
        cout << ch;
    }

    return 0;
}
```

Folosirea funcţiilor membre tip pentru streamuri bazate pe matrice

Streamurile bazate pe matrice pot, de asemenea, să fie accesibile prin intermediul funcţiilor membre standard din **ios**, cum ar fi **get()** şi **put()**. Starea streamului bazat pe matrice poate fi determinată cu funcţii ca **rdstate()**, **good()**, **bad()** ş.a.m.d. De asemenea, puteţi folosi **eof()** pentru a determina când se ajunge la sfârşitul matricei. De exemplu, următorul program determină cum se citeşte conţinutul unei matrice folosindu-se **get()**.

```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char s[] = "abcdefghijklmnop";

    istrstream ins();

    char ch;

    // Aceasta va citi continutul oricarui tip de matrice.
    while (!ins.eof()) {
        ins.get(ch);
        cout << ch;
    }

    return 0;
}
```


Funcţiile membre din clasa **ios** sunt utile în special atunci când aveţi nevoie să citiţi sau să scrieţi buffere de date.

Streamuri de intrare/ieşire bazate pe matrice

Pentru a crea un stream bazat pe matrice care efectuează atât intrări cât şi ieşiri, folosiţi această funcţie constructor tip **strstream**:

strstream iostr(char **buffer*, int *marime*, int *mod*);

Aici, *buffer* indică spre un şir care va fi folosit pentru operaţii de I/O. Valoarea din *marime* specifică mărimea matricei. Valoarea din *mod* determină cum operează streamul. Pentru operaţii de I/O normale, aceasta va fi **ios::in | ios::out**. Pentru intrare, matricea va trebui să se încheie cu null.

Iată un program care foloseşte o matrice pentru a efectua atât intrări cât şi ieşiri:

```
// Efectueaza atat intrari cat si iesiri.
#include <iostream.h>
#include <strstrea .h>

main()
{
    char iostr[80];

    strstream ios(iostr, sizeof(iostr), ios::in | ios::out);

    int a, b;
    char sir[80];
```

```
    ios << "10 20 testare";
    ios >> a >> b >> sir;
    cout << a << " " << b << " " << sir << endl;

    return 0;
}
```

Pentru început, el scrie **10 20 testare** în **iostr** şi apoi citeşte această informaţie din **iostr**.

Accesul aleator în cadrul matricelor

Amintiţi-vă că tuturor operaţiilor de I/O bazate pe matrice li se aplică acţiunile de I/O normale, deci şi accesul aleator folosind **seekg()** şi **seekp()**. De exemplu, următorul program caută al optulea caracter în matricea **iostr** şi îl afişează. (El va afişa **h**.)

```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char iostr[80];

    strstream ios(iostr, sizeof(iostr), ios::in |ios::out);

    char ch;
    ios << "abcdefghijklmnopqrstuvwxyz";
    ios.seekg(7, ios::beg);
    ios >> ch;
    cout << "Caracterul in pozitia 7: " << ch;

    return 0;
}
```

Puteţi să căutaţi oriunde *în interiorul* matricei de I/O; nu însă şi dincolo de limitele sale.

Streamurilor bazate pe matrice li se pot aplica, de asemenea, funcţii ca **tellg()** şi **tellp()**.

Utilizarea matricelor dinamice

În prima parte a acestui capitol, când aţi legat un stream de o matrice de ieşire, matricea şi mărimea sa au fost transmise constructorului **ostrstream**. Această

caracteristică este bună atât timp cât cunoaşteţi numărul maxim de caractere pe care îl veţi pune în acea matrice. Totuşi, ce se întâmplă dacă nu ştiţi cât de mare trebuie să fie matricea de ieşire? Soluţia problemei este să folosiţi a doua formă a constructorului **ostrstream**, prezentată aici:

```
ostrstream();
```

Când este folosit acest constructor, **ostrstream** creează şi întreţine o matrice alocată dinamic. Acestei matrice îi este permis să crească în lungime pentru a corespunde ieşirii pe care trebuie să o stocheze.

Reţineţi că funcţia constructor **ostrstream** *nu* returnează efectiv un pointer către matricea alocată. Pentru a avea acces dinamic la o astfel de matrice, trebuie să folosiţi o a doua funcţie numită **str()**. Ea „îngheaţă" matricea şi returnează un pointer către ea. O dată ce o matrice dinamică este îngheţată, nu mai poate fi folosită pentru o nouă ieşire. De aceea, nu îngheţaţi matricea înainte să terminaţi de transmis caractere.

Iată un program care foloseşte o matrice de ieşire dinamică:

```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char *p;

    ostrstream outs; // matrice alocata dinamic

    outs << "Imi place C++ " ";
    outs << -10 << hex << " ";
    outs.setf(ios::showbase);
    outs << 100 << ends;

    p = outs.str(); /* Ingheata bufferul dinamic si
                       returneaza un pointer spre el. */
    cout << p;

    delete p; // Elibereaza bufferul dinamic creat de
              // ostrstream().
    return 0;
}
```

Aşa cum ilustrează acest program, o dată ce matricea dinamică a fost îngheţată, vă revine responsabilitatea să eliberaţi memoria pentru sistem când aţi

terminat cu ea. Dacă nu aţi îngheţat deloc matricea, memoria este eliberată automat când este distrus streamul.

Puteţi, de asemenea, să folosiţi matricele dinamice de I/O cu clasa **strstream**, care poate să efectueze atât intrări cât şi ieşiri cu o matrice. Pentru a crea o matrice dinamică folosind **strstream**, folosiţi această construcţie:

strstream();

Aceasta va crea o matrice dinamică ce va fi capabilă să creeze intrări şi ieşiri.

Manipulatori şi operaţii de I/O bazate pe matrice

Deoarece streamurile bazate pe matrice sunt aceleaşi ca şi oricare alt stream, manipulatorii pe care îi creaţi în general pentru I/O pot fi folosiţi fără nici o modificare şi de operaţiile de I/O bazate pe matrice. De exemplu, în **Capitolul 17**, au fost creaţi manipulatorii de ieşire **sd()** şi **ss()** (săgeată la dreapta şi, respectiv, săgeată la stânga) pentru I/O de la consolă. Următorul program arată că ei sunt la fel de eficienţi şi pentru operaţii de I/O bazate pe matrice.

```
// Acest program foloseste manipulatorii creati de
// utilizator pentru operatii de I/O bazate pe matrice
#include <strstrea .h>
#include <iostream.h>

// Sageata la dreapta
ostream &sd(ostream &stream)
{
    stream << "-------> ";
    return stream;
}

// Sageata la stanga
ostream &ss(ostream &stream)
{
    stream << " <-------";
    return stream;
}

main()
{
    char sir[80];

    ostream outs(sir, sizeof(sir));

    outs << sd << "Priviti acest numar: ";
    outs << 1000000 << ss << ends; // terminat cu null

    cout << " " << sir;

    return 0;
}
```

Acest program afişează următoarea ieşire:

```
-------> Priviti acest numar: 1000000 <-------
```

Funcţii create de utilizator pentru extragere şi inserţie

Deoarece streamurile bazate pe matrice sunt ca oricare alte streamuri, puteţi să creaţi propriile funcţii de extragere şi de inserţie, în acelaşi fel în care le creaţi pentru alte tipuri de streamuri. De exemplu, următorul program construieşte o clasă numită **punct**, care memorează coordonatele x, y ale unui punct în spaţiul bidimensional. Funcţia de inserţie supraîncărcată pentru această clasă afişează un mic plan de coordonate şi indică poziţia punctului. Pentru simplificare, mărimea coordonatelor x, y este restrânsă la intervalul dintre 0 şi 5.

```
#include <iostream.h>
#include <strstrea .h>

const int marime = 5;

class punct {
    int x, y;
public:
    punct(int i, int j) {
    // pentru acest exemplu se limiteaza x si y la
    // intervalul dintre 0 si marime
        if(i>marime) i = marime; if (i<0) i=0;
        if(j>marime) j = marime; if (j<0) j=0;
        x=i; y=j;
    }

    // O functie de insertie pentru punct.
```

```
    friend ostream &operator<<(ostream &stream, punct O);
};

ostream &operator<<(ostream &stream, punct O)
{
    register int i, j;

    for(j=marime; j>=0; j--) {
          stream << j;
          if(j == o.y) {
                for(i=0; i<O.x; i++) stream << " ";
                stream << '*';
          }
          stream << "\n";
    }

    for(i=0; i<=marime; i++) stream << " " << i;
    stream << "\n";

    return stream;
}

main()
{
    punct a(2, 3), b(1, 1);
    char sir[200];
    // afiseaza folosind mai intai cout
    cout << "Afiseaza folosind cout:\n";
    cout << a << "\n" << b << "\n\n";
    // acum foloseste operatii de I/O bazate pe RAM
    ostream outs(sir, sizeof(sir));

    // acum afiseaza outs folosind si formatarea in RAM
    outs << a << b << ends;

    cout << "Afiseaza folosind formatarea in RAM:\n";
    cout << sir;

    return 0;
}
```

Acest program determină următoarea ieşire:


```
Afiseaza folosind cout:
5
4
3   *
2
1
0
 0 1 2 3 4 5
5
4
3
2
1  *
0
 0 1 2 3 4 5

Afiseaza folosind formatarea in RAM:
5
4
3   *
2
1
0
 0 1 2 3 4 5
5
4
3
2
1  *
0
 0 1 2 3 4 5
```

Utilizări ale formatării bazate pe matrice

În C, funcţiile **sprintf()** şi **sscanf()** sunt folosite în special pentru pregătirea ieşirilor către sau citirii intrărilor de la echipamente nestandardizate. Însă, datorită capacităţilor lui C++ de a supraîncărca funcţiile de inserţie şi de extragere relativ la clase şi de a crea manipulatori adaptaţi, pot fi manevrate uşor multe echipamente exotice. Aceasta face ca necesitatea formatării în RAM să fie mai puţin importantă. Totuşi, există încă multe utilizări pentru operaţii de I/O bazate pe matrice.

O utilizare curentă a unei formatări bazate pe matrice este să construiţi un şir care să fie folosit ca intrare ori pentru o bibliotecă standard ori pentru o terţă

funcţie. De exemplu, puteţi să aveţi nevoie să construiţi un şir care să fie analizat de funcţia de bibliotecă standard **strtok()**. (Funcţia **strok()** „analizează" - adică descompune în elemente - un şir.) Un alt loc în care pot fi utilizate operaţiile de I/O bazate pe matrice este în editoarele de texte care efectuează operaţii complexe de formatare. Deseori, este mai uşor să folosiţi operaţiile de I/O din C++ bazate pe matrice pentru a construi un şir complex, decât s-o faceţi manual. Foarte utilă în programarea în Windows este şi construirea şirurilor folosind formatarea bazată pe matrice. Windows nu conţine nici o funcţie standard care să asigure ieşiri formatate într-o fereastră, ci trebuie să construiţi din timp toate ieşirile formatate.

Capitolul 20

Şabloane

O caracteristică relativ nouă în C++ este *şablonul* (template). Cu un şablon este posibil să creaţi *funcţii generice* şi *clase generice*. Într-o funcţie sau clasă generică, tipul de date asupra căruia operează acestea este specificat ca un parametru. De aceea, puteţi folosi o funcţie sau o clasă cu mai multe tipuri de date diferite, fără să rescrieţi versiunile specifice acestora. Aici sunt discutate atât funcţiile cât şi clasele generice.

NOTĂ: *Şabloanele nu au făcut parte dintre specificaţiile originale ale limbajului C++, dar au fost adăugate în 1990. Ele sunt definite de către standardul ANSI C++ propus şi sunt admise de majoritatea compilatoarelor de C++ disponibile astăzi.*

Funcţii generice

O funcţie generică defineşte un set general de operaţii care vor fi aplicate unor tipuri de date variate. Unei astfel de funcţii tipul de date asupra căruia va opera îi este transmis ca parametru. Utilizând acest mecanism, poate fi aplicată aceeaşi procedură unui domeniu larg de date. După cum probabil ştiţi, mulţi algoritmi au aceeaşi logică, indiferent de tipul de date asupra căruia operează. De exemplu, algoritmul de sortare Quicksort este acelaşi chiar dacă se aplică unei matrice de întregi sau uneia de tip **float**. Ceea ce diferă este doar tipul de date care este sortat. Creând o funcţie generică, puteţi defini natura algoritmului, independent de date. O dată făcut acest lucru, atunci când se execută funcţia, compilatorul generează automat codul corect pentru tipul de date folosit efectiv. În esenţă, când creaţi o funcţie generică, creaţi o funcţie care se supraîncarcă singură, automat.

O astfel de funcţie este creată cu ajutorul cuvântului cheie **template**. Semnificaţia normală a cuvântului „şablon" reflectă exact utilizarea sa în C++. El este folosit pentru a crea un şablon (tipar, model) care descrie ce va face o funcţie, lăsând compilatorul să completeze detaliile necesare. Iată forma generală a unei definiţii de funcţie de tip **template**:

template <class *Tip*> *tip-retur nume-func(lista parametri)*
{
// corpul funcţiei
}

Tip este un nume care ţine locul tipului de date folosite de către funcţie. Acest nume poate fi folosit în cadrul definirii unei funcţii. Dar, el ţine doar un loc pe care compilatorul îl va înlocui automat cu tipul de date efectiv, atunci când va crea o versiune specifică a funcţiei.

Următorul exemplu scurt creează o funcţie generică ce inversează între ele valorile celor două variabile cu care este apelată. Deoarece procesul general de înlocuire a două valori este independent de tipul acestora, este foarte bine să îl descrieţi într-o funcţie generică.


```
// Exemplu de functie sablon.
#include <iostream.h>
// Aceasta este o functie sablon.
template <class X> void inloc(X  &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}

main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "i, j originali: " << i << ' ' << j << endl;
    cout << "x, y originali: " << x << ' ' << y << endl;
    cout << "a, b originali: " << a << ' ' << b << endl;
    inloc(i,j); // inverseaza intregii
    inloc(x, y); // inverseaza float
    inloc(a,b); // inverseaza caractere
    cout << "i, j inversati: " << i << ' ' << j << endl;
    cout << "x, y inversati: " << x << ' ' << y << endl;
    cout << "a, b inversati: " << a << ' ' << b << endl;
    return 0;
}
```

Să privim cu atenţie acest program. Linia:

```
template <class X> void inloc(X &a, X &b)
```

spune compilatorului două lucruri: că este creat un şablon şi că va urma definirea generică. **X** este aici un tip generic care este folosit ca substitut. După zona **template** este declarată funcţia **inloc()**, folosind pe **X** ca tip de date pentru valorile care vor fi inversat. În **main()**, funcţia **inloc()** este apelată cu trei tipuri de date: **int**, **float** şi **char**. Deoarece **inloc()** este o funcţie generică, compilatorul îi va crea automat trei versiuni - una care va inversa valorile întregi, una care va inversa pe cele în virgulă mobilă şi una care va inversa caractere.

Iată câţiva alţi termeni care sunt folosiţi uneori atunci când vine vorba despre şabloane şi pe care îi puteţi întâlni în literatura de C++. Mai întâi, o funcţie generică (adică o definire a unei funcţii precedate de declararea **template**) este numită şi *funcţie şablon*. Când compilatorul creează o versiune concretă a acestei funcţii se spune că a creat o *funcţie generată*. Procesul de generare a unei funcţii este numit *de exemplificare* (*de instanţiere*). Altfel spus, o funcţie generată este un exemplar specific al unei funcţii şablon.

Practic, zona **template** a definirii unei funcţii generice nu trebuie să fie pe aceeaşi linie cu numele funcţiei. De exemplu, următorul fragment este, de asemenea, un mod uzual de a scrie funcţia **inloc()**.

```
template <class X>
void inloc(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

Dacă folosiţi această formă, este important să înţelegeţi că între instrucţiunea **template** şi începutul funcţiei generice nu poate să apară nici o altă instrucţiune. De exemplu, fragmentul prezentat mai jos nu va fi compilat.

```
// Acesta nu va fi compilat
template <class X>
int i; // aceasta este greseala
void inloc(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

După cum arată comentariul, specificarea **template** trebuie să fie imediat urmată de definiţia funcţiei.

O funcţie cu două tipuri generice

Într-o instrucţiune **template** puteţi să definiţi mai mult de un tip generic, folosind o listă separată prin virgulă. De exemplu, următorul program creează o funcţie generică cu două tipuri generice.


```
#include <iostream.h>

template <class tip1, class tip2>
void funcmea(tip1 x, tip2 y)
{
    cout << x << ' ' << y << endl;
}
main()
{
    funcmea(10, "hi");

    funcmea(0.23, 12L);

    return 0;
}
```

În acest exemplu, substituenţii **tip1** şi **tip2** sunt înlocuiţi de către compilator cu tipurile de date **int** şi **char***, respectiv **double** şi **long**, atunci când generează exemplarele specifice pentru **funcmea()** din cadrul funcţiei **main()**.

REŢINEŢI: Când creaţi o funcţie generică, permiteţi, de fapt, compilatorului să genereze atâtea versiuni ale funcţiei câte sunt necesare pentru a trata modurile diferite în care funcţia este apelată de către programul dvs.

Supraîncărcarea explicită a unei funcţii generice

Chiar dacă o funcţie şablon se supraîncarcă singură când este necesar, puteţi să o supraîncărcaţi şi explicit. Dacă supraîncărcaţi o funcţie generică, atunci ea suprascrie (sau „ascunde") funcţia generică relativ la acea versiune specifică. Să luăm, de exemplu, această variantă a primului exemplu:

```
// Suprascrierea unei functii sablon.
#include <iostream.h>

template <class X> void inloc(X  &a, X &b)
{
    X temp;

    temp = a;
    a = b;
```

```
    b = temp;
}

// aceasta suprascrie versiunea generica a inloc().
void inloc(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "In functia inloc(int &, int &) suprascrisa.\n";
}

main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "i, j originali: " << i << ' ' << j << endl;
    cout << "x, y originali: " << x << ' ' << y << endl;
    cout << "a, b originali: " << a << ' ' << b << endl;

    inloc(i,j); // aceasta apeleaza inloc()supraincarcat
                // explicit
    inloc(x, y); // inverseaza float
    inloc(a,b); // inverseaza caractere
    cout << "i, j inversate: " << i << ' ' << j << endl;
    cout << "x, y inversate: " << x << ' ' << y << endl;
    cout << "a, b inversate: " << a << ' ' << b << endl;
    return 0;
}
```

După cum menţionează comentariile, la apelarea funcţiei **inloc(i, j)** este invocată versiunea sa redefinită explicit în program. Astfel, compilatorul nu generează această variantă a funcţiei, deoarece funcţia generică pierde prioritatea în faţa celei supraîncărcate explicit.

Supraîncărcarea manuală a şablonului, aşa cum s-a arătat în acest exemplu, vă permite să scrieţi o versiune a funcţiei generice adaptată la o situaţie specială. Totuşi, în general, dacă aveţi nevoie de mai multe versiuni ale unei funcţii pentru diferite tipuri de date, ar trebui să folosiţi, mai degrabă, şablonul şi nu supraîncărcarea.

Restricţii pentru funcţia generică

Funcţiile generice sunt similare celor supraîncărcate, doar că ele sunt mai restrictive. Când este supraîncărcată o funcţie, în interiorul fiecărei variante se pot efectua acţiuni diferite. Dar o funcţie generică trebuie să efectueze aceeaşi acţiune pentru toate versiunile - doar tipul de date poate să difere. De exemplu, în programul următor, funcţiile supraîncărcate nu pot fi înlocuite de funcţii generice deoarece ele nu efectuează aceleaşi lucruri.

```
#include <iostream.h>
#include <math.h>

void funcmea(int i)
{
    cout << "valoarea este: " << i << \n";
}

void funcmea(double d)
{
    double parteintrg;
    double partefrac;

    partefrac = modf(d, &parteintrg);
    cout << "Partea fractionara: " << partefrac;
    cout << "\n";
    cout << "Partea intreaga: " << partintrg;
}

main()
{
    funcmea(1);
    funcmea(12.2);

    return 0;
}
```

Iată alte câteva restricţii ale funcţiilor şablon. O funcţie virtuală nu poate fi funcţie şablon. Destructorii nu pot fi şabloane. O funcţie şablon trebuie să folosească editoarele de legături din C++. (Aceasta înseamnă că ea nu poate folosi o specificare de editare de legături. Vedeţi **Capitolul 22.**)

Aplicarea funcţiilor generice

Funcţiile generice reprezintă una dintre cele mai folositoare caracteristici ale limbajuluilui C++. Ele pot fi aplicate oricăror situaţii. Cum s-a menţionat mai devreme, oricând aveţi o funcţie care defineşte un algoritm general, puteţi să-l introduceţi într-o funcţie generică. O dată ce aţi făcut aşa ceva, puteţi să o folosiţi cu orice tip de date, fără să mai fie necesar să o rescrieţi. Înainte de a continua cu clasele generice, vor fi date două exemple de aplicare a funcţiilor generice. Ele ilustrează cât de uşor este să profitaţi de această facilitate puternică din C++.

O sortare generică

Sortarea este tipul de operaţie pentru care sunt destinate funcţiile generice. Într-o mare măsură un algoritm de sortare este acelaşi, indiferent de tipul de date care se sortează. Următorul program ilustrează aceasta, creând o sortare generică prin amestecare. Deşi sortarea prin amestecare este un algoritm puţin eficient, modul său de operare este clar şi evident şi oferă un exemplu uşor de înţeles. (Poate că veţi dori să încercaţi să creaţi o versiune generică pentru algoritmul dvs. de sortare preferat.) Funcţia **amestec()** va sorta orice tip de matrice. Ea este apelată cu un pointer către primul ei element şi cu numărul ei de elemete.

```
// Un caz de sortare prin amestecare.
#include <iostream.h>
template <class X> void amestec(
   X *elemente, // pointer spre matricea de sortat
   int numar) // numarul de elemente din matrice
{
   register int a, b;
   X t;

   for(a=1; a<numar; a++)
         for(b=numar-1; b>=a; b--)
               if(elemente[b-1] > elemente[b]) {
                  // inverseaza elementele
                  t = elemente[b-1];
                  elemente[b-1] = elemente[b];
                  elemente[b] = t;
               }
   }

main()
{
   int imatrice[7] = {7, 5, 4, 3, 9, 8, 6};
```



```
   double dmatrice[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

   int i;

   count << "Iata matricea de intregi nesortata: ";
   for(i=0; i<7; i++)
         cout << imatrice[i] << ' ';
   cout << endl;
   cout << "Iata matricea de double nesortata: ";

   for(i=0; i<5; i++)
         cout << dmatrice[i] << ' ';
   cout << endl;

   amestec(imatrice, 7);
   amestec(dmatrice, 5);

   cout << "Iata matricea de intregi sortata: ";
   for(i=0; i<7; i++)
         cout << imatrice[i]; << ' ';
   cout << endl;
   cout << "Iata matricea de double sortata: ";
   for(i=0; i<5; i++)
         cout << dmatrice[i] << ' ';
   cout << endl;

   return 0;
}
```

Acest program generează următoarea ieşire:

```
Iata matricea de intregi nesortata: 7 5 4 3 9 8 6
Iata matricea de double nesortata: 4.3 2.5 -0.9 100.2 3
Iata matricea de intregi sortata: 3 4 5 6 7 8 9
Iata matricea de double sortata: -0.9 2.5 3 4.3 100.2
```

După cum puteţi vedea, programul precedent creează două matrice: una pentru întregi şi una pentru **double**. Apoi le sortează pe fiecare. Deoarece **amestec()** este o funcţie şablon, ea este supraîncărcată automat pentru a se adapta celor două tipuri de date diferite.

Compactarea unei matrice

O altă funcţie care beneficiază de crearea sa ca funcţie şablon este numită **compact()**. Ea compactează elementele unei matrice. După cum ştiţi probabil din

experienţa de programare, este uzual să doriţi să ştergeţi nişte elemente de undeva dintr-o matrice şi apoi să deplasaţi celelalte elemente, astfel încât spaţiile nefolosite să ajungă la sfârşit. Acest fel de operaţie este acelaşi pentru toate tipurile de matrice deoarece el este independent de tipul de date asupra căruia lucrează efectiv. Funcţia generică **compact()** prezentată în programul următor este apelată cu un pointer spre primul element al matricei, numărul de elemente din aceasta şi cu primul şi ultimul indice al elementelor care trebuie extrase. Apoi funcţia le şterge şi compactează matricea. În scop demonstrativ, ea pune zero în locul elementelor nefolosite de la sfârşitul matricei care au fost eliberate prin compactare.

```
// O functie generic de compactare a unei matrice.
#include <iostream.h>

template <class X> void compact(
    X *elemente, // pointer spre matricea ce trebuie compactata
    int numar, // numar de elemente din matrice
    int start, // indicele de pornire pentru zona compactata
    int stop) // indicele de incheiere a zonei compactate
{
    register int i;

    for(i=stop+1; i<numar; i++, start++)
        elemente[start] = elemente[i];
    /* Pentru demonstratie, restul matricei va fi
       completata cu 0. */
    for( ; start<numar; start++) element[start] = (X) 0;
}

main()
{
    int num[7] = {0, 1, 2, 3, 4, 5, 6};
    char sir[18] = "Functii generice";

    int i;

    cout << "Iata matricea de intregi, necompactata: ";
    for(i=0; i<7; i++)
        cout << num[i] << ' ';
    cout << endl;

    cout << "Iata sirul necompactat: ";
    for(i=0; i<17; i++)
```



```
        cout << sir[i] << ' ';
    cout << endl;

    compact(num, 7, 2, 4);
    compact(sir, 17, 6, 10);
    cout << "Iata matricea de intregi compactata: ";
    for(i=0; i<7; i++)
        cout << num[i] << ' ';
    cout << endl;

    cout << "Iata sirul compactat: ";
    for(i=0; i<17; i++)
        cout << sir[i] << ' ';
    cout << endl;
    return 0;

}
```

Acest program compactează două tipuri de matrice. Una este o matrice de întregi, iar cealaltă este un şir. Dar, funcţia **compact()** va avea efect pentru oricare tip de matrice. Iată ieşirea acestui program:

```
Iata matricea de intregi necompactata: 0 1 2 3 4 5 6
Iata sirul necompactat: F u n c t i i   g e n e r i c e
Iata matricea de intregi compactata: 0 1 5 6 0 0 0
Iata sirul compactat: F u n c t i e r i c e
```

După cum ilustrează exemplul precedent, o dată ce începeţi să gândiţi în termeni de şabloane, vă vor veni spontan în minte multe utilizări. Atâta vreme cât logica de bază a unei funcţii este independentă de date, ea poate fi creată ca funcţie generică.

Clase generice

În afară de funcţiile generice, puteţi să definiţi, de asemenea, o clasă generică. Când faceţi asta, creaţi o clasă care defineşte toţi algoritmii folosiţi de ea, dar tipul de date care este manevrat efectiv va fi specificat ca un parametru la crearea obiectelor acelei clase.

Clasele generice sunt folositoare când o clasă conţine caracteristici generale. De exemplu, acelaşi algoritm care tratează o înşiruire de întregi va lucra şi pentru o înşiruire de caractere. De asemenea, acelaşi mecanism care întreţine liste înlănţuite pentru adrese poştale va gestiona şi liste pentru articole auto. Folosind o clasă generică, puteţi să definiţi operaţiile care vor controla pentru orice tip de date

înşiruirea, listele înlănţuite ş.a.m.d. Compilatorul va genera automat tipul corect al obiectului, bazat pe tipul pe care îl specificaţi atunci când este creat acesta.

Iată forma generală a declarării unei clase generice:

template<class *Tip*>class *nume-clasa* {
.
.
.
}

Aici, *Tip* ţine locul numelui tipului pe care îl veţi specifica când se defineşte un exemplar al clasei. Dacă este necesar, puteţi să definiţi mai mult de un tip de date generice, folosind o listă separată prin virgule.

O dată ce aţi construit o clasă generică, puteţi crea un anumit exemplar al acesteia, folosind forma generală:

nume-clasa<tip>ob;

Aici, *tip* este numele tipului de date cu care va opera clasa. Funcţiile membre ale claselor generice sunt automat şi ele însele generice.

În programul următor, clasa **stiva** (folosită prima dată în **Capitolul 11**) este rescrisă ca o clasă generică. Astfel, ea poate fi folosită pentru a asigura o memorie stivă pentru orice tip de obiect. În exemplul prezentat aici sunt create o stivă de caractere, una de întregi şi una de numere în virgulă mobilă.

```
// Prezinta o clasa generica pentru memoria stiva.
#include <iostream.h>

const int SIZE = 100;

// Aceasta creeaza o clasa generica pentru stiva.
template <class STip> class stiva {
    STip stv[SIZE];
    int vis;
public:
    stiva();
    ~stiva();
    void pune(STip i);
    STip scoate();
};

// functia constructor pentru stiva
template <class STip > stiva<STip>::stiva()
```



```
{
    vis = 0;
    cout << "Stiva este initializata\n";
}
/* Functia destructor a stivei
   Aceasta functie nu este necesara. Ea este inclusa
   doar pentru demonstratie. */
template <class STip> stiva<STip>::~stiva()
{
    cout << "Stiva este distrusa\n";
}

// Pune un obiect in stiva.
template <class STip> void stiva<STip>::pune(STip i)
{
    if(vis==SIZE) {
          cout << "Stiva este plina.";
          return;
    }
    stv[vis] = i;
    vis++;
}

// Scoate un obiect din stiva.
template <class STip> STip stiva<STip>::scoate()
{
    if(vis==0) {
          cout << "Stiva este vida.";
          return 0;
    }
    vis--
    return stv[vis];
    }

main()
{
    stiva<int> a; // creeaza stiva de intregi
    stiva<double> b; // creeaza stiva pentru double
    stiva<char> c; // creeaza stiva pentru caractere

    int i;

    // foloseste stivele pentru intregi si pentru double
```

```
    a.pune(1);
    b.pune(99.3);
    a.pune(2);
    b.pune(-12.23);

    cout << a.scoate() << " ";
    cout << a.scoate() << " ";
    cout << b.scoate() << " ";
    cout << b.scoate() << "\n";

    // prezinta stiva pentru caractere
    for(i=0; i<10; i++) c.pune((char) 'A'+i);
    for(i=0; i<10; i++) cout << c.scoate();
    cout << "\n";

    return 0;
}
```

După cum puteți vedea, declararea clasei generice este similară cu cea a unei funcții generice. Tipul de date generic este folosit în declarația clasei și a funcțiilor membre. Până când nu se declară nici un obiect al stivei nu se determină tipul de date efectiv. Când este declarat un anumit exemplar pentru **stiva**, compilatorul generează automat toate funcțiile și variabilele necesare tratării tipului de date efectiv. În acest exemplu, sunt declarate trei tipuri de stivă (una pentru întregi, una pentru **double** și una pentru caractere). Fiți atenți, în special, la aceste declarații:

```
stiva<int> a; // creeaza stiva de intregi
stiva<double> b; // creeaza stiva pentru double
stiva<char> c; // creeaza stiva pentru caractere
```

Observați cum este transmis tipul de date dorit în interiorul parantezelor unghiulare. Puteți să modificați tipul de date memorat de stivă, schimbând tipul de date specificate atunci când sunt create obiectele de tip **stiva**. De exemplu, folosind următoarea declarare, ați fi putut crea o altă stivă care să fi conținut pointeri pentru caractere.

```
stiva<char *> carpointstiv;
```

Puteți crea, de asemenea, stive pentru a memora tipurile de date pe care le creați. De exemplu, dacă doriți să memorați informații despre adrese, folosiți această structură:


```
struct adr {
    char nume[40];
    char strada[40];
    char oras[30];
    char judet[3];
    char zip[12];
}
```

Apoi, pentru a folosi **stiva** ca să generați o stivă care să memoreze obiecte de tip **adr**, folosiți o declarație ca aceasta:

```
stiva<adr> obiect;
```

După cum ilustrează clasa **stiva**, funcțiile și clasele generice asigură un instrument puternic pe care îl puteți folosi pentru a obține cât mai mult de la programele dvs. deoarece vă permit să definiți forma generală a unui obiect care poate fi folosit apoi cu oricare tip de date. Sunteți astfel salvați de plictiseala de a construi implementări separate pentru fiecare tip de date cu care doriți să lucreze clasa respectivă. Compilatorul creează automat, pentru dvs., versiunea specifică a clasei.

Un exemplu cu două tipuri de date generice

O clasă șablon poate avea mai mult decât un singur tip de date generic. Declarați pur și simplu toate tipurile de date necesare clasei într-o listă separată prin virgule, în cadrul specificației **template**. De exemplu, următorul program scurt creează o clasă care folosește două tipuri de date generice.

```
/* Acest exemplu foloseste doua tipuri de date generice
   intr-o definire de clasa.
*/
#include <iostream.h>

template <class Tip1, class Tip2> class clasamea
{
    Tip1 i;
    Tip2 j;
public:
    clasamea(Tip1 a, Tip2  b) {i = a; j = b; }
    void arata() { cout << i << ' ' << j << '\n'; }
};

main()
```

```
{
    clasamea<int, double> ob1(10, 0.23);
    clasa mea<char, char *> ob2('X', "Acesta este un test");

    ob1.arata(); // arata int, double
    ob2.arata(); // arata char, char *

    return 0;
}
```

Acest program are următoarea ieşire:

```
10 0.23
X Acesta este un test
```

Programul declară două tipuri de obiecte. **ob1** foloseşte întregi, iar **ob2** foloseşte caractere şi pointeri pentru caractere. În ambele cazuri compilatorul generează automat datele şi funcţiile corespunzătoare pentru a se adapta felului în care s-a creat obiectul.

Crearea unei clase generice de matrice

Să urmărim o aplicaţie obişnuită a unei clase generice. După cum aţi văzut în Capitolul 14, puteţi să supraîncărcaţi operatorul [ ]. Făcând asta, puteţi crea propriile implementări de matrice. Aceasta vă permite crearea de „matrice sigure", care vă asigură verificarea limitelor în timpul rulării. După cum ştiţi, este posibil ca în C++ să depăşiţi (sau să fiţi sub) limita unei matrice în timpul rulării, fără să vă apară un mesaj de eroare. În schimb, dacă stabiliţi o clasă care conţine matricea şi permite accesul la ea doar prin operatorul pentru indice [ ] supraîncărcat, atunci puteţi să depistaţi un indice în afara limitelor.

Combinând supraîncărcarea operatorului cu o clasă generică, este posibil să creaţi un tip de matrice generică sigură, ce poate fi folosită pentru a crea matrice sigure de orice tip de date, după cum se arată în următorul program.

```
// Un exemplu de matrice generica sigura.
#include <iostream.h>
#include "stdlib.h"

const int SIZE = 10;

template <class ATip> class atip {
    ATip a[SIZE];
public:
```



```
    atip() {
        register int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    ATip &operator[](int i);
};
// Asigura verificarea limitelor pentru atip.
template <class ATip> ATip &atip<ATip>::operator[](int i)
{
    if(i<0; || i> SIZE-1) {
        cout << "\nValoarea de ";
        cout << i << " a indicelui este in afara limitei.\n";
        exit(1);
    }

    return a[i];
}

main()
{
    atip<int> intob; // matrice de intregi
    atip<double> doubleob; // matrice de double

    int i;

    cout << "Matrice de intregi: ";
    for(i=0; i<SIZE; i++) intob[i] = i;
    for(i=0; i<SIZE; i++) cout << intob[i] << " ";
    cout << '\n';

    cout << "Matrice de double: ":
    cout.precision(2);
    for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
    for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
    cout << "\n';

    intob[12] = 100; // genereaza eroare in timpul rularii

    return 0;
}
```

Acest program implementează un tip de matrice generică sigură şi apoi demonstrează utilizarea sa creând o matrice de întregi şi una de **double**. (Puteţi să

încercați să creați și alte tipuri de matrice.) După cum se arată în acest exemplu, o parte din puterea claselor generice este aceea că ele vă permit să scrieți codul o dată, să îl depanați și apoi să îl aplicați oricărui tip de date fără să îl rescrieți pentru fiecare aplicație.

Pentru simplificare, programul anterior folosește matrice cu mărime fixă. Puteți, însă, să schimbați clasa **atip** astfel încât să poată fi declarate matrice de dimensiuni variabile. Pentru a realiza aceasta, specificați dimensiunile matricei ca parametru al funcției constructor **atip** și alocați matricea dinamic.

NOTĂ: *În Capitolul 25 veți găsi un alt exemplu de clasă generică. Ea creează o listă dublu înlănțuită capabilă să memoreze orice tip de obiect.*

Capitolul 21

Tratarea excepțiilor

Acest capitol discută modul de *tratare a excepțiilor* în C++. *Tratarea excepțiilor* vă permite să rezolvați într-un mod ordonat erorile din timpul rulării; cu ajutorul acestui mecanism din C++, atunci când apare o eroare în timpul rulării, programul dvs. poate să apeleze automat o rutină de rezolvare a acesteia. Principalul avantaj al tratării excepțiilor este acela că el automatizează mult din codul de remediere a erorilor care, mai înainte, trebuia scris „de mână" în programele mai mari.

NOTĂ: Tratarea excepțiilor nu a făcut parte din specificația originală de C++, ci s-a dezvoltat din 1984 până în 1989, iar astăzi este definită în standardul ANSI C++ propus și este admisă de majoritatea compilatoarelor disponibile.

Bazele tratării excepțiilor

Tratarea excepțiilor în C++ este construită pe trei cuvinte-cheie: **try**, **catch** și **throw**. În termeni generali, instrucțiunile din program pe care doriți să le urmăriți în căutarea excepțiilor sunt conținute într-un bloc **try**. Dacă o excepție (ceea ce înseamnă o eroare) apare în blocul **try**, ea este „lansată" (folosindu-se **throw**), după care este „prinsă" cu instrucțiunea **catch** și prelucrată. Următoarea discuție se bazează pe această descriere generală.

După cum am spus, orice instrucțiune care lansează o excepție trebuie să fie executată dintr-un bloc **try**. (Și funcțiile apelate din interiorul blocului **try** pot, de asemenea, să lanseze o excepție.) Orice excepție trebuie să fie captată de instrucțiunea **catch**, ce urmează imediat după instrucțiunea **try**, care va lansa excepția. Iată forma generală pentru **try** și **catch**:

```
try {
      // bloc try
}
catch(tip1 arg) {
      // bloc catch
}
catch (tip2 arg) {
      // bloc catch
}
catch (tip3 arg) {
      // bloc catch
}
.
.
.
catch (tipN arg) {
      // bloc catch
}
```


Blocul **try** trebuie să conțină acea secțiune a programului în care doriți să căutați erorile. Ea se poate întinde de la câteva instrucțiuni dintr-o funcție până la întregul corp al funcției **main()** dintr-un bloc **try** (ceea ce determină urmărirea efectivă a întregului program).

Când este lansată o excepție, ea este captată de instrucțiunea **catch** corespunzătoare, care o prelucrează. Pot fi mai multe instrucțiuni **catch** asociate uneia **try**. Care dintre instrucțiunile **catch** este folosită, se determină prin tipul excepției. Cu alte cuvinte, dacă tipul de date specificat de o ramură **catch** corespunde cu cel al excepției, atunci se execută acea instrucțiune **catch** (iar toate celelalte sunt ignorate). Când este prinsă o excepție, valoarea sa va fi memorată de *arg*. Pot fi captate orice tipuri de date, inclusiv clasele pe care le creați. Dacă nu se găsește nici o excepție (deci că nu apare nici o eroare în blocul **try**), atunci nu se va executa nici o instrucțiune **catch**.

Iată forma generală a instrucțiunii **throw**:

throw *excepție*;

throw trebuie să fie executată ori chiar din interiorul blocului **try**, ori din interiorul oricărei funcții apelate (direct sau indirect) din blocul **try**. *excepție* este valoarea lansată.

Dacă lansați o excepție pentru care nu există nici o instrucțiune tip **catch** aplicabilă, poate să apară o încheiere anormală a programului. Dacă aveți un compilator ce respectă standardul ANSI C++ propus, atunci lansarea unei excepții netratate determină apelarea funcției **terminate()**. Implicit, **terminate()** apelează **abort()** pentru a opri programul dvs., dar puteți specifica, dacă doriți, propriul modul de încheiere. Pentru detalii, va trebui să consultați specificațiile bibliotecii compilatorului.

Iată un exemplu simplu care arată felul în care operează tratarea excepțiilor în C++.

```
// Un exemplu simplu pentru tratarea exceptiilor.
#include <iostream.h>

main()
{
   cout << "Start\n";

   try { // inceputul blocului try
         cout << "In interiorul blocului try\n";
         throw 100; // lanseaza o eroare
         cout << "Aceasta nu se va executa";
```

```
    }
    catch (int i) { // preia eroarea
          cout << "Am prins o exceptie -- valoarea este: ";
          cout << i << "\n";
    }

    cout << "End";

    return 0;
}
```

Acest program va afişa următoarea ieşire:

```
Start
In interiorul blocului try
Am gasit o exceptie -- valoarea este: 100
End
```

Urmăriţi cu atenţie programul precedent. După cum puteţi vedea, există un bloc **try** care conţine trei instrucţiuni şi o ramură **catch(int i)** care prelucrează excepţia **int**. În interiorul blocului **try** vor fi executate doar două din cele trei instrucţiuni: prima instrucţiune **cout** şi instrucţiunea **throw**. O dată ce a fost găsită o excepţie, controlul trece la expresia **catch**, iar blocul **try** se încheie. Aceasta *nu* înseamnă că este apelat **catch**, ci că s-a executat un transfer al programului în acest punct (stiva programului s-a modificat automat ca atare. Astfel, instrucţiunea **cout** ce urmează după **throw** nu se va executa niciodată.

De obicei, codul din cadrul unei instrucţiuni **catch** încearcă să remedieze eroarea printr-o anumită acţiune. Dacă eroarea poate fi corectată, execuţia va continua cu instrucţiunile ce urmează după **catch**. Dar, de multe ori, o eroare nu poate fi reparată, iar blocul **catch** va termina programul cu un apel la **exit()** sau la **abort()**.

După cum s-a menţionat, tipul excepţiei trebuie să corespundă celui specificat în instrucţiunea **catch**. De exemplu, în programul precedent, dacă schimbaţi tipul din instrucţiunea **catch** în **double**, atunci excepţia nu va mai fi prinsă şi va apărea o terminare anormală a programului. Iată această modificare:

```
// Acest exemplu nu va functiona.
#include <iostream.h>

main()
{
    cout << "Start\n";
    try { // inceputul blocului try
```



```
          cout << "In interiorul blocului try\n";
          throw 100; // lanseaza o eroare
          cout << "Aceasta nu se va executa";
    }
    catch (double i) { // Nu va lucra pentru o exceptie int
          cout << "Am gasit o exceptie -- valoarea este: ";
          cout << i << "\n";
    }

    cout << "End"

    return 0;
}
```

Acest program va produce următoarea ieşire, deoarece excepţia de tip **int** nu va fi prinsă de către instrucţiunea **catch(double i)**.

```
Start
In interiorul blocului try
Încheiere anormala a programului (Abnormal program termination)
```

O excepţie poate fi lansată dintr-o instrucţiune care este în exteriorul blocului **try** atât timp cât se află într-o funcţie care este apelată din interiorul blocului **try**. De exemplu, acesta este un program valid:

```
/* Lansarea unei exceptii dintr-o functie din exteriorul
   blocului try
*/
#include <iostream.h>

void Xtest (int test)
{
   cout << "In interiorul lui Xtest, testul este: "
        << test << "\n";
   if(test) throw test;
}

main()
{
    cout << "Start\n";
    try { // inceputul blocului try
          cout << "In interiorul blocului try\n";
          Xtest(0);
```

```
            Xtest(1);
            Xtest(2);
      }
      catch (int i) { // preia o eroare
            cout << "Am gasit o exceptie -- valoarea este: ";
            cout << i << "\n";
      }

      cout << "Sfarsit";

      return 0;
}
```

Acest program va determina următoarea ieşire:

```
Start
In interiorul blocului try
In interiorul lui Xtest, testul este: 0
In interiorul lui Xtest, testul este: 1
Am gasit o exceptie -- valoarea este: 1
Sfirsit
```

Un bloc **try** poate fi localizat într-o funcţie. Când se întâmplă aşa ceva, mecanismul de tratare a erorilor referitor la funcţie este reiniţializat de câte ori este apelată aceasta. Să examinăm, de exemplu, acest program:

```
#include <iostream.h>
// Perechea try/catch poate exista si in alta functie decat
// main().
void Xmanip(int test)
{
    try{
          if(test) throw test;
    }
    catch(int i) {
          cout << "A gasit exceptia #:"<< i << '\n';
    }
}

main()
{
    cout << "Start\n";
```



```
      Xmanip(1);
      Xmanip(2);
      Xmanip(0);
      Xmanip(3);

      cout << "Sfarsit";

      return 0;
}
```

Acest program afişează următoarea ieşire:

```
Start
A gasit exceptia #: 1
A gasit exceptia #: 2
A gasit exceptia #: 3
Sfarsit
```

După cum puteţi vedea, sunt lansate trei excepţii. După fiecare excepţie, funcţia se returnează. Când aceasta este apelată din nou, se reiniţializează tratarea excepţiilor.

Este important de înţeles că un cod asociat cu instrucţiunea **catch** va fi executat doar dacă va capta o **excepţie**, iar altfel va fi ignorat. Execuţia nu va ajunge niciodată pe cale normală la instrucţiunea **catch**.) De exemplu, în următorul program nu este lansată nici o excepţie, astfel încât instrucţiunea **catch** nu se execută.

```
#include <iostream.h>

main()
{
    cout << "Start\n";
    try { // inceputul unui bloc try
          cout << "In interiorul blocului try\n";
          //    throw 100; // acest throw nu se executa
          cout << "Tot in blocul try\n";
    }
    catch (int i) { // gaseste o eroare
          cout << "A gasit o eroare -- valoarea este: ";
          cout << i << "\n";
    }

    cout << "Sfirsit";
```

```
    return 0;
}
```

Acest program determină următoarea ieşire:

```
Start
In interiorul blocului try
Tot in interiorul blocului try
Sfarsit
```

După cum puteţi vedea, instrucţiunea **catch** este ignorată de către execuţie.

Folosirea instrucţiunilor catch multiple

După cum am spus, unui **try** îi puteţi asocia mai mult de o instrucţiune **catch**. De fapt, chiar se obişnuieşte. Însă, fiecare **catch** trebuie să capteze un tip diferit de excepţie. De exemplu, acest program prinde atât întregi cât şi şiruri:

```
#include <iostream.h>

// Pot fi captate diferite tipuri de exceptii.
void Xmanip(int rest)
{
   try{
      if(test) throw test;
      else throw "Valoarea este zero";
   }
   catch(int i) {
      cout << "A preluat exceptia #: " << i << '\n';
   }
   catch(char *sir) {
      cout << "A preluat un sir: ";
      cout << sir << '\n';
   }
}

main()
{

   cout << "Start\n";
   Xmanip(1);
   Xmanip(2);
```



```
   Xmanip(0);
   Xmanip(3);

   cout << "Sfarsit";

   return 0;
}
```

Acest program determină următoarea ieşire:

```
Start
A preluat exceptia #: 1
A preluat exceptia #: 2
A preluat un sir: Valoarea este zero
A preluat exceptia #: 3
Sfarsit
```

După cum puteţi vedea, fiecare instrucţiune **catch** răspunde doar tipului său.

Expresiile **catch** sunt verificate, în general, în ordinea în care apar în program. Se execută doar instrucţiunea care corespunde excepţiei, toate celelalte blocuri **catch** fiind ignorate.

Opţiuni de tratare a excepţiilor

În C++ există multe caracteristici şi nuanţe suplimentare pivind tratarea excepţiilor, care fac limbajul mai uşor şi mai convenabil de folosit. Ele sunt discutate în continuare.

Preluarea tuturor excepţiilor

În anumite circumstanţe veţi dori ca un manipulator al excepţiilor să le capteze pe toate, şi nu doar pe cele de un anumit tip. Aceast lucru este uşor de realizat. Folosiţi pur şi simplu această formă pentru **catch**:

```
catch(...) {
   // procesează toate excepţiile
}
```

Cele trei puncte corespund aici tuturor tipurilor de date.

Următorul program ilustrează **catch(...)**.

```
// Acest exemplu captează toate exceptiile.
#include <iostream.h>

void Xmanip(int test)
{
    try{
          if(test==0) throw test; // lanseaza int
          if(test==1) throw 'a'; // lanseaza char
          if(test==2) throw 123.23; // lanseaza double
    }
    catch(...) { // preia toate exceptiile
          cout << "Am prins una!\n";
    }
}

main()
{
    cout << "Start\n";

    Xmanip(0);
    Xmanip(1);
    Xmanip(2);

    cout << "Sfarsit";

    return 0;
}
```

Acest program afişează următoarea ieşire:

```
Start
Am prins una!
Am prins una!
Am prins una!
Sfarsit
```

După cum puteţi vedea, toate cele trei **throw** au fost prinse printr-o singură instrucţiune **catch**.

O utilizare foarte bună pentru **catch(...)** este ca un ultim **catch** pentru o serie de **catch**. În această poziţie asigură o captare implicită utilă sau o instrucţiune de tip „prinde tot". De exemplu, această versiune uşor diferită a programului precedent preia explicit excepţii de tip întreg, dar se bazează pe **catch(...)** pentru a le capta pe celelalte.


```
// Acest exemplu foloseste catch(...) ca varianta implicita.
#include <iostream.h>

void Xmanip(int test)
{
    try{
          if(test==0) throw test; // lanseaza int
          if(test==1) throw 'a'; // lanseaza char
          if(test==2) throw 123.23; // lanseaza double
    }
    catch(int i) { // preia o exceptie int
          cout << "Am prins un intreg\n";
    catch(...) { // preia toate celelalte exceptii
          cout << "Am prins una!\n";
    }
}

main()
{
    cout << "Start\n";

    Xmanip(0);
    Xmanip(1);
    Xmanip(2);

    cout << "Sfarsit";

    return 0;
}
```

Iată ieşirea produsă de acest program:

```
Start
Am prins un intreg
Am prins una!
Am prins una!
Sfarsit
```

După cum sugerează exemplul, utilizarea instrucţiunii **catch(...)** implicite este o modalitate bună de a capta toate excepţiile pe care nu doriţi să le trataţi explicit. De asemenea, preluând toate excepţiile, evitaţi ca o excepţie netratată să determine o încheiere anormală a programului.

Restricţii pentru excepţii

Când o funcţie este apelată dintr-un bloc **try**, puteţi să restrâgeţi tipul de excepţie pe care îl va lansa ea. De fapt, puteţi preveni, de asemenea, ca funcţia să lanseze orice fel de excepţie. Pentru a realiza aceste restricţii, trebuie să adăugaţi definirii funcţiei o clauză **throw**. Forma generală a acesteia este prezentată aici:

```
tip-returnat nume-func(lista-arg)throw(lista-tip)
{
    //...
}
```

Aici pot fi lansate de către funcţie doar acele tipuri conţinute în *lista-tip*, listă separată prin virgule. Lansarea oricăror alte tipuri de expresii va determina o încheiere anormală a programului. Dacă nu doriţi ca o funcţie să fie capabilă să elimine *nici* o expresie, atunci folosiţi o listă goală.

Dacă aveţi un compilator care respectă standardul ANSI C++ propus, atunci încercarea de a lansa o excepţie care nu este admisă de către funcţie va determina apelarea funcţiei **unexpected()**. Implicit, aceasta apelează funcţia **abort()**, care duce la terminarea anormală a programului. Totuşi, dacă doriţi, puteţi specifica propriul dvs. manipulator pentru încheiere. Pentru detalii va trebui să consultaţi biblioteca de referinţă a compilatorului.

Următorul program arată cum se restrâng tipurile de excepţii ce pot fi lansate dintr-o funcţie.

```
// Restrangerea tipurilor lansate de functie.
#include <iostream.h>

// Aceasta functie poate lansa doar int, char si double.
void Xmanip(int test) throw(int, char, double)
{
    if(test==0) throw test; // lanseaza test
    if(test==1) throw 'a'; // lanseaza char
    if(test==2) throw 123.23; // lanseaza double
}

main()
{
    cout << "Start\n";

try{
    Xmanip(0); // incercati, de asemenea, sa pasati 1 si 2
               // functiei Xmanip()
}
catch(int i) {
    cout << "Am prins un intreg\n";
}
catch(char c) {
    cout << "Am prins char\n";
}
catch(double d) {
    cout << "Am prins double\n";
}

cout << "Sfarsit";

return 0;
}
```


În acest program funcţia **Xmanip()** poate să lanseze doar expresii de tip întreg, caracter şi **double**. Dacă va încerca să lanseze orice alt tip de excepţie, va produce o terminare anormală a programului. (Va fi apelată funcţia **unexpected()**.) Pentru a vedea un asemenea exemplu, îndepărtaţi **int** din lista de argumente şi rulaţi programul din nou.

Este important să înţelegeţi că o funcţie poate fi limitată doar în ceea ce priveşte tipurile de excepţii pe care le lansează înapoi în blocul **try** care a apelat-o. Dar un bloc **try** *din interiorul* unei funcţii poate să lanseze orice excepţie, atât timp cât acesta este captat în *interiorul* funcţiei respective. Restricţia se aplică doar când se lansează o excepţie în afara funcţiei.

Următoarea modificare a funcţiei **Xmanip()** o împiedică să mai lanseze vreo excepţie.

```
// Aceasta functie NU poate lansa nici o exceptie!
void Xmanip(int test) throw()
{
/* Urmatoarele instructiuni nu mai lucreaza,
   ele vor duce la o terminare anormala a programului. */
    if(test==0 throw test;
    if(test==1) throw 'a';
    if(test==2) throw 123.23;
}
```

Relansarea unei excepţii

Dacă doriţi să relansaţi o excepţie din cadrul unui manipulator de excepţii, puteţi să o faceţi apelând **throw** fără nici o excepţie. Aceasta va determina ca excepţia

curentă să fie transmisă unei secvențe **try/catch** exterioare. Cel mai bun motiv pentru a face acest lucru este să permiteți manipulatorilor multipli accesul la acele excepții. De exemplu, poate că un manipulator de excepții tratează un aspect al uneia, iar al doilea se ocupă de altul. O excepție poate fi relansată doar din blocul **catch** (sau din orice funcție apelată din interiorul acestuia). Când relansați o excepție, ea nu va fi preluată de aceeași instrucțiune **catch**, ci se va deplasa spre următoarea. Programul de mai jos ilustrează relansarea unei excepții, și anume cea de tipul **char***.

```
// Exemplu de "relansare" a unei exceptii.
#include <iostream.h>

void Xmanip()
{
    try{
          throw "hello"; // elimina un char *
    }
    catch(char *) { // capteaza un char *
          cout << "Am prins un char * in interiorul
                    functiei Xmanip\n";
          throw ; // relanseaza char * in afara functiei
    }
}

main()
{
    cout << "Start\n";

    try{
          Xmanip();
    }
    catch(char *) {
          cout << "Am prins un char * in interiorul
                    functiei main\n";
    }

    cout << "Sfarsit";

    return 0;
}
```

Acest program afișează următoarea ieșire:


```
Start
Am prins un char * in interiorul lui Xmanip
Am prins un char * in interiorul lui main
Sfarsit
```

Aplicații ale tratării excepțiilor

Tratarea excepțiilor este destinată asigurării unei modalități structurate prin care programul dvs. să reacționeze la evenimente neobișnuite. Aceasta înseamnă că, atunci când apare o eroare, manipulatorul să facă ceva rațional. De exemplu, să considerăm următorul program simplu, care introduce două numere și împarte pe primul la al doilea. El folosește tratarea erorilor pentru a preveni o eroare de împărțire la zero.

```
#include <iostream.h>

void impart(double a, double b);

main()
{
    double i, j;
    do {
          cout << "Introduceti deimpartitul (0 pentru stop): ";
          cin >> i;
          cout << "Introduceti impartitorul: ";
          cin >> j;
          impart(i, j);
    }     while(i != 0);

    return 0;
}
void impart(double a, double b)
{
    try {
          if(!b) throw b; // verifica impartirea la 0
          cout << "Rezultat: " << a/b << endl;
    }
    catch (double b) {
          cout << "Nu pot imparti la zero.\n";
    }
}
```

Deşi programul precedent este foarte simplu, el ilustrează esenţa tratării excepţiilor. De vreme ce împărţirea la zero nu este permisă, programul nu poate continua dacă al doilea număr introdus este zero. În acest caz, excepţia este tratată prin neefectuarea împărţirii (care ar fi determinat o terminare anormală a programului) şi prin avertizarea utilizatorului asupra erorii. Programul solicită apoi din nou introducerea a două numere. Astfel, eroarea a fost tratată corect, iar utilizatorul poate să continue lucrul cu programul. Aceeaşi concepţie va sta şi la baza aplicaţiilor mai complexe pentru tratarea erorilor.

Tratarea erorilor este folositoare, în special, pentru a ieşi, atunci când apare o eroare catastrofală, dintr-un lanţ de rutine imbricate profund. În această privinţă, tratarea excepţiilor în C++ este proiectată pentru a înlocui funcţiile destul de greoaie din C, **setjump()** şi **longjump()**.

REŢINEŢI: Motivul principal pentru utilizarea tratării excepţiilor este de a asigura o modalitate ordonată de tratare a erorilor. Aceasta presupune, dacă este posibil, corectarea situaţiei.

Capitolul 22

Elemente diverse şi caracteristici avansate

Acest capitol prezintă câteva apecte din C++ care nu au fost examinate în altă parte a acestei cărţi. Caracteristicile includ funcţiile de conversie, constructorii de copii, argumentele implicite ale funcţiilor, specificaţii pentru editarea legăturilor, şi alte elemente noi, adăugate de standardul propus pentru ANSI C++ sau diferenţe între C şi C++.

Argumente implicite pentru funcţii

C++ permite ca, atunci când nu se specifică un argument corespunzător pentru un parametru al unei funcţii, să i se atribuie acelui parametru o valoare implicită. Aceasta este specificată într-o manieră similară sintactic iniţializării unei variable.

Următorul exemplu declară **funcmea()** ca preluând un argument în virgulă mobilă, de tip **double**, cu valoarea implicită 0.0.

```
void funcmea(double d = 0.0)
{
    .
    .
    .
}
```

Acum, **funcmea()** poate fi apelată în două moduri, aşa cum arată următorul exemplu.

```
funcmea(198.234); // transmite explicit o valoare
funcmea(); // lasa functia sa foloseasca valoarea implicita
```

Prima apelare pasează lui **d** valoarea 198.234. Cea de-a doua îi dă automat valoarea implicită 0.

Un motiv pentru care argumentele implicite sunt incluse în C++ este acela că ele oferă programatorului o altă metodă pentru a trata programele de complexitate mai mare. Pentru a face faţă varietăţii mari de situaţii, o funcţie conţine destul de frecvent mai mulţi parametri decât sunt necesari pentru o utilizare normală. De aceea, când se aplică argumentele implicite, trebuie să vă amintiţi şi să specificaţi doar argumentele care sunt semnificative pentru cele mai uzuale situaţii, nu şi pentru cazul cel mai general. De exemplu, multe dintre funcţiile de I/O din C++ descrise în capitolele precedente folosesc argumente implicite exact din acest motiv.

O ilustrare simplă pentru cât de folositor poate fi un argument implicit al unei funcţii este realizată de către funcţia **stergecr()** din următorul program. Această funcţie şterge ecranul generând o serie de caractere de salt la linie nouă (nu este modul cel mai eficient, dar este suficient pentru acest exemplu). Deoarece un monitor obişnuit afişează 25 de linii de text, este furnizat argumentul implicit 25.

Dar, pentru că unele terminale pot să afişeze mai mult sau mai puţin de 25 de linii (de multe ori depinzând de modul video în care se lucrează), puteţi să supraîncărcaţi argumentul implicit specificând unul în mod explicit.

```
#include <iostream.h>

void stergecr(int marime=25);

main()
{
    register int i;

    for(i=0; i<30; i++) cout << i << endl;
    cin.get();
    stergecr(); // curata 25 de linii

    for(i=0; i<30; i++) cout << i << endl;
    cin.get()
    strergecr(10); // sterge 10 linii

    return 0;
}

void stergecr(int marime)
{
    for(; marime; marime--) cout << endl;
}
```

După cum ilustrează programul, nu este necesară specificarea nici unui argument la apelarea lui **stergecr()** atunci când valoarea implicită corespunde situaţiei. Este însă posibil ca, atunci când este necesar, să neglijaţi valoarea implicită şi să daţi o valoare diferită parametrului **marime**.

Un argument implicit poate fi folosit, de asemenea, ca indicator care spune funcţiei să reia un argument anterior. Pentru a ilustra acest tip de utilizare este prezentată o funcţie scurtă numită **intra()**, care indentează automat un şir cu un număr specificat. Iată, pentru început, o versiune a acestei funcţii care nu foloseşte un argument implicit:

```
void intra(char *sir, int aliniat)
{
    if(aliniat < 0) aliniat = 0;

    for( ; aliniat; aliniat--) cout << " ";
```

```
    cout << sir << "\n";
}
```

Această versiune a funcției **intra()** este apelată având ca prim argument șirul care va fi afișat și ca al doilea argument mărimea indentării. Deși forma funcției nu este cu nimic greșită, puteți să îi măriți utilitatea furnizându-i un argument implicit pentru parametrul **aliniat** care îi spune lui **intra()** să folosească indentarea specificată anterior. Este destul de uzuală afișarea unui bloc de text cu fiecare linie indentată la aceeași distanță. În această situație, în loc să fie necesar să transmiteți în mod repetat același argument pentru **aliniat**, puteți să îi dați o valoare implicită care îi spune lui **intra()** să indenteze ca mai înainte. Această abordare este ilustrată în programul următor:

```
#include <iostream.h>

/* Indentare implicita cu -1. Aceasta valoare ii spune
   functiei sa refoloseasca valoarea anterioara. */
void intra(char *sir, int aliniat = -1);

main()
{
    intra("Va salut", 10);
    intra("Acesta va fi indentat, implicit, cu 10 spatii");
    intra("Acesta va fi indentat cu 5 spatii", 5);
    intra("Acesta nu va fi indentat", 0;

    return 0;
}

void intra(char *sir, int aliniat)
{
    static i = 0; // pastreaza valoarea anterioara a
                  // aliniatului

    if(aliniat>=0)
          i = aliniat;
    else // refoloseste valoarea vechiului aliniat
          aliniat = i;

    for( ; aliniat; aliniat--) cout << " ";

    cout << sir << "\n";
}
```


Acest program afișează următoarea ieșire:

```
          Va salut
          Acesta va fi indentat, implicit, cu 10 spatii
     Acesta va fi indentat cu 5 spatii
Acesta nu va fi indentat
```

Când creați funcții care au valori implicite pentru argumente, este important să vă amintiți că aceste valori trebuie specificate doar o dată, și anume prima dată când este declarată funcția într-un fișier. În exemplul precedent argumentul implicit a fost menționat în prototipul funcției **intra()**. Dacă veți încerca să specificați o nouă valoare implicită (sau chiar aceeași) în definirea ei, compilatorul va afișa o eroare și nu va compila programul. Deși nu pot fi redefinite argumente implicite ale aceleiași funcții, puteți să specificați argumente implicite diferite pentru fiecare versiune a unei funcții supraîncărcate.

Toți parametrii care preiau valori implicite trebuie să apară la dreapta celor care nu preiau. De exemplu, este incorect să definiți astfel **intra()**:

```
// gresit!
void intra(int aliniat = -1, char *sir);
```

O dată ce ați definit parametrii care preiau valori implicite, nu puteți să specificați un parametru care nu este implicit. Deci, o declarație ca aceasta este, de asemenea, greșită, și nu va fi compilată:

```
int funcmea(float f, char *sir, int i=10, int j);
```

Deoarece lui **i** i s-a dat o valoare implicită, atunci și lui **j** trebuie să i se dea una.

Puteți folosi parametri impliciți într-o funcție constructor a unui obiect. De exemplu, clasa **cub** prezentată aici păstrează dimensiunile unui cub ca valori întregi. Funcția sa constructor inițializează implicit toate dimensiunile cu zero dacă nu se asigură nici un alt argument, așa cum este prezentat mai jos:

```
#include <iostream.h>

class cub {
    int x, y, z;
public:
    cub(int i=0, int j=0, int k=0) {
          x=i;
          y=j;
          z=k;
    }
```

```
    int volum() {
          return x*y*z;
    }
};

main()
{
    cub a(2,3,4), b;

    cout << a.volum() << endl;
    cout << b.volum();

    return 0;
}
```

Includerea argumentelor implicite într-o funcţie constructor, atunci când este cazul, prezintă două avantaje. În primul rând, vă scuteşte să mai creaţi şi un constructor cu lista de parametri vidă. De exemplu, dacă parametrii funcţiei **cub()** nu ar fi fost impliciţi, ar fi fost nevoie şi de al doilea constructor, prezentat aici, care să trateze declararea lui **b** (care nu specifică nici un argument):

```
cub() {x=0; y=0; z=0}
```

În al doilea rând, introducerea valorilor iniţiale uzuale este mai convenabilă decât specificarea lor de fiecare dată când este declarat un obiect.

Utilizarea corectă a argumentelor implicite

Chiar dacă argumentele implicite pot fi un instrument puternic când sunt folosite corect, ele pot fi şi sursă de erori. Scopul principal al argumentelor implicite este să permită funcţiei să lucreze eficient, într-un mod uşor de utilizat, fără a-şi pierde însă din flexibilitate. Pentru aceasta, toate argumentele implicite ar trebui să se încadreze în varianta în care este folosită funcţia în majoritatea timpului. De exemplu, un argument implicit îşi are rostul dacă este folosit 90% din timp. Dar, dacă o valoare va fi întâlnită doar în 10% din apelări, iar în restul timpului argumentul care corespunde acestui parametru va varia mult, nu este recomandabil să stabiliţi un argument implicit. Rostul acestora este să furnizeze valorile pe care programatorul le asociază cu o anumită funcţie. Când nu există doar o singură valoare asociată curent unui parametru, nu există nici un motiv pentru crearea unui argument implicit. De altfel, declararea unor argumente implicite de care nu este realmente nevoie, va distruge structura codului creat de dvs. - va deruta şi va induce în eroare pe cei ce vor citi programul. Este, desigur,

subiectiv cât să alegeţi între 10% şi 90% pentru ca argumentele implicite să devină cu adevărat utile, dar 51% pare o limită rezonabilă.

Un alt factor important pe care ar trebui să-l urmăriţi când folosiţi argumente implicite este ca nici un astfel de argument să nu determine o acţiune dăunătoare sau distructivă. Deci, utilizarea accidentală a unui argument implicit nu va determina o catastrofă.

Argumente implicite sau supraîncărcare?

Înainte de a părăsi subiectul argumentelor implicite, va fi discutată o altă aplicaţie. În unele situaţii, aceste argumente pot fi folosite ca o formă rapidă de supraîncărcare a funcţiei. Pentru a vedea de ce, imaginaţi-vă că doriţi să creaţi două versiuni proprii ale funcţiei standard **strcat()**. Prima versiune va lucra exact ca **strcat()** şi va adăuga întreg conţinutul unui şir la sfârşitul altuia. Cea de-a doua va prelua un al treilea argument care va specifica numărul de caractere de concatenare. A doua versiune va adăuga, deci, doar un anumit număr de caractere dintr-un şir la sfârşitul altuia. De aceea, presupunând că denumiţi funcţiile dvs. **strcatmeu()**, ele vor avea următoarele prototipuri:

```
void strcatmeu(char *s1, char *s2, int lung);
void strcatmeu(char *s1, char *s2);
```

Prima versiune va copia **lung** caractere din **s2** la sfârşitul lui **s1**. A doua versiune va copia întregul şir spre care indică **s2** la sfârşitul şirului spre care indică **s1** şi va lucra precum **strcat()**.

Deşi nu este greşit să implementaţi două versiuni ale funcţiei **strcatmeu()** pentru a realiza cele două efecte pe care le urmăriţi, există o cale mai simplă. Folosind un argument implicit, puteţi să creaţi doar o singură versiune a funcţiei **strcatmeu()** care să lucreze în ambele moduri. Următorul program prezintă acest procedeu.

```
// O versiune proprie pentru strcat().
#include <iostream.h>
#include <string.h>

void strcatmeu(char *s1, char *s2, int lung = 0);

main()
{
   char sir1[80] = "Acesta este un test";
   char sir2[80] = "0123456789";

   strcatmeu(sir1, sir2, 5); // concateneaza 5 caractere
```

```
    cout << sirl << 'n\';

    strcpy(sirl, "Acesta este un test"); // reinitializeaza sirl

    strcatmeu(sirl, sir2); // concateneaza intregul sir
    cout << sirl << '\n';

    return 0;
}
// O versiune proprie pentru strcat().
void strcatmeu(char *s1, char *s2, int lung)
{
    // gaseste sfarsitul sirului s1
    while(*s1) s1++;

    if(lung==0) lung = strlen(s2);

    while(*s2 && lung) {
          *s1 = *s2; // copiaza caractere
          s1++;
          s2++;
          lung--;
    }

    *s1 = '\0'; // terminare cu null pentru s1
}
```

Aici, **strcatmeu()** adaugă până la **lung** caractere din şirul spre care indică **s2** la sfârşitul şirului spre care indică **s1**. Însă, dacă **lung** este zero (ca în cazurile în care se păstrează valoarea implicită), **strcatmeu()** concatenează întregul şir spre care indică **s2** la sfârşitul lui **s1**. (De aceea, când **lung** este zero, funcţia operează ca şi funcţia **strcat()** standard.) Folosind argumentele implicite pentru **lung**, este posibil să combinaţi ambele operaţii într-o singură funcţie. În acest fel, argumentele implicite asigură uneori o formă rapidă de supraîncărcare a funcţiilor.

Crearea funcţiilor de conversie

În unele situaţii aveţi nevoie să folosiţi un obiect dintr-o clasă într-o expresie care implică alte tipuri de date. Uneori funcţiile operator supraîncărcate pot să asigure calea de a face aceasta. Dar, în alte cazuri, tot ceea ce doriţi este o simplă conversie a tipului clasei în tipul destinaţie. Pentru a trata acest caz, C++ vă permite să creaţi *funcţii proprii de conversie*. O funcţie de conversie transformă

tipul clasei într-un tip compatibil cu cel al restului expresiei. Forma generală a unei funcţii de conversie a tipului este:

operator *tip*() {return *valoare*;}

tip este aici tipul ţintă în care convertiţi clasa dvs., iar *valoare* este valoarea obiectului după conversie. Funcţiile de conversie returnează date de tipul *tip* şi nu este permis nici un alt specificator de tip pentru valoarea întoarsă. De asemenea, nu pot apărea parametri. O astfel de funcţie trebuie să fie un membru al clasei pentru care este definită. Funcţiile de conversie sunt moştenite şi pot fi virtuale.

Următoarea ilustrare a modului de creare a unei funcţii de conversie foloseşte clasa **stiva** prezentată în **Capitolul 11**. Să presupunem că doriţi să puteţi combina obiecte de tipul **stiva** cu o expresie de tip întreg. Mai mult, să presupunem că valoarea unui obiect din **stiva** folosită într-o expresie de tip întreg este numărul de valori aflate curent în memoria stivă. (Puteţi să doriţi să faceţi ceva de genul acesta dacă, de exemplu, folosiţi obiecte de tip **stiva** într-o simulare şi vreţi să urmăriţi cât de repede se umple stiva.) O cale de rezolvare este să convertiţi un obiect de tipul **stiva** într-un întreg care reprezintă numărul de elemente din memoria stivă. Pentru a realiza acest lucru, folosiţi o funcţie de conversie care arată astfel:

```
operator int() {return vis;}
```

Iată un program care ilustrează cum lucrează funcţiile de conversie:

```
#include <iostream.h>

const int SIZE=100;

// aceasta creeaza clasa stiva
class stiva {
    int stiv[SIZE];
    int vis;
public:
    stiva() {vis=0;}
    void pune(int i);
    int scoate(void);
    operator int() {return vis;} // conversia lui stiva in int
};

void stiva::pune(int i)
{
    if(vis==SIZE) {
```

```
            cout << "Stiva este plina.";
            return;
        }
        stiv[vis] = i;
        vis++;
}

int stiva::scoate()
{
    if(vis==0) {
            cout << "Stiva este vida.";
            return 0;
    }
    vis--;
    return stiv[vis];
}

main()
{
    stiva stiv;
    int i, j;

    for(i=0; i<20; i++) stiv.pune(i);

    j = stiv; // converteste in intreg

    cout << j << " elemente in stiva\n";

    cout << SIZE - stiv << " spatii libere\n";

    return 0;
}
```

Acest program afişează următoarea ieşire:

```
20 elemente in stiva
80 spatii libere
```

După cum ilustrează programul, când un obiect de tip **stiva** este folosit într-o expresie de tip întreg, aşa cum este **j = stiv**, obiectului i se aplică funcţia de conversie. În acest caz particular, funcţia returnează valoarea 20. De asemenea, funcţia este apelată când **stiv** este scăzut din **SIZE**.

Iată alt exemplu de funcţie de conversie. Acest program creează o clasă numită

putere(), care memorează şi calculează rezultatul unor numere ridicate la o putere. Ea obţine rezultatul în virgulă mobilă. Puteţi să folosiţi obiecte de tipul **putere()** în expresii care implică alte valori tip **double** asigurând o funcţie de conversie în tipul **double** şi returnând rezultatul.

```
#include <iostream.h>

class putere {
    double b;
    int e;
    double val;
public:
    putere(double baza, int exp);
    putere operator+(putere o) {
            double baza;
            int exp;
            baza = b +o.b;
            exp = e + o.e;

            putere temp(baza, exp);
            return temp;
    }
    operator double() {return val;} // conversie in double
};

putere::putere(double baza, int exp)
{
    b = baza;
    e = exp;
    val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) val = val * b;
}

main()
{
    putere x(4.0, 2);
    double a;

    a = x; // conversie in double
    cout << x + 100.2; // converteste x in double si aduna 100.2
    cout << "\n";
```

```
    putere y(3.3, 3), z(0, 0);

    z = x + y; // nici o conversie
    a = z; // conversie in double
    cout << a;

    return 0;
}
```

După cum puteţi vedea, când **x** este utilizat în expresia **x + 100.2**, este folosită funcţia de conversie pentru a determina o valoare de tip **double**. Reţineţi, de asemenea, că în expresia **x + y** nu se aplică nici o conversie deoarece ea implică doar obiecte de tip **putere**.

După cum puteţi deduce din exemplele anterioare, există multe situaţii în care este bine să creaţi o funcţie de conversie pentru o clasă. Deseori aceste funcţii asigură o sintaxă mai naturală când obiectele claselor sunt combinate cu tipuri încorporate. În particular, în cazul clasei **putere()**, posibilitatea conversiei în **double** face ca obiectele din acea clasă să fie „normale", mai uşor atât de programat cât şi de înţeles în expresiile matematice „normale".

Puteţi crea funcţii de conversie distincte pentru a preîntâmpina diverse cerinţe. Aţi putea defini una care converteşte, de exemplu, în **long**. Fiecare va fi aplicată automat în funcţie de tipul expresiei.

Constructori pentru copii de obiecte

Implicit, când un obiect este folosit pentru a iniţializa pe un altul, C++ efectuează o copie pe biţi. Aceasta înseamnă că este creată o copie identică a obiectului iniţial în obiectul ţintă. Deşi metoda este adecvată perfect în multe cazuri - şi în general rezultatul este exact cel pe care îl doriţi - există situaţii în care copia pe biţi nu poate fi folosită. Una dintre cele mai obişnuite situaţii în care trebuie să evitaţi o astfel de copie este atunci când unui obiect i se alocă memorie la creare. De exemplu, să luăm două obiecte, A şi B, din aceeaşi clasă numită *TipClasa*, cărora li se alocă memorie în momentul creării. De asemenea, să presupunem că A există deja. Deci A are deja alocată memorie. Mai departe, să presupunem că A este folosit pentru a iniţializa pe B, aşa cum se arată aici:

TipClasa B = A;

Dacă se face o simplă copie pe biţi, atunci B va fi o copie exactă a lui A. Aceasta înseamnă că B va avea alocată exact aceeaşi zonă de memorie pe care o foloseşte A, în loc să i se aloce propria sa zonă de memorie. Este clar că nu acesta este rezultatul dorit. De exemplu, dacă *TipClasa* conţine un destructor care eliberează memoria, atunci aceeaşi zonă de memorie va fi eliberată de două ori, când A şi B sunt distruse!

Aceeaşi problemă poate să apară în încă două cazuri. Primul este atunci când este făcută o copie a unui obiect pentru a fi pasată ca argument către o funcţie. Al doilea este atunci când se creează un obiect temporar ca valoare returnată de către o funcţie. (Amintiţi-vă că obiectele temporare sunt create automat pentru a păstra valoarea întoarsă de o funcţie şi că ele mai pot fi construite în anumite alte circumstanţe.)

Pentru a rezolva problema, C++ vă permite să creaţi un *constructor de copii*, pe care îl foloseşte compilatorul atunci când un obiect este utilizat pentru a iniţializa un altul. Când există un constructor pentru copie, se renunţă la copierea pe biţi. Forma generală a unui constructor pentru copie este:

numeclasa(const *numeclasa &o*) {
    // corpul constructorului
}

Aici *o* este o referinţă spre obiectul din partea dreaptă a iniţializării. Este permis unui constructor de copie să aibă parametri în plus, atâta timp cât ei au definite argumente implicite. Însă, în toate cazurile, primul parametru trebuie să fie o referinţă spre obiectul care face iniţializarea.

Iniţializarea apare în trei feluri: când un obiect iniţializează altul, când se face o copie a unui obiect pentru a fi transmisă unei funcţii sau când este generat un obiect temporar (cel mai adesea ca valoare returnată). De exemplu, fiecare dintre următoarele instrucţiuni implică iniţializare:

```
clasamea x = y; // initializare
func(x); // transmitere de parametru
y = func(); // memoreaza obiectul temporar
```

În continuare se dă un exemplu în care este necesară o funcţie constructor de copie explicită. Programul creează un tip de matrice „sigură" de întregi, (foarte) limitat, care verifică limitele pentru a nu fi depăşite. Memoria pentru fiecare matrice este alocată prin utilizarea operatorului **new**, iar în interiorul fiecărui obiect - matrice este păstrat un pointer spre aceasta.

```
/* Acest program creeaza o clasa de matrice "sigure".
   Deoarece spatiul pentru matrice este alocat folosind
   new, pentru a aloca memorie cand un obiect-matrice este
   folosit pentru a initializa un altul, este introdus un
   constructor de copie.
*/
#include <iostream.h>
```

```
#include <stdlib.h>

class matrice {
    int *p;
    int marime;
public:
    matrice(int mar) {
          p = new int[mar];
          if(!p) exit(1);
          marime = mar;
    }
    ~matrice() {delete [] p;}

    // constructor de copie
    matrice(const matrice &a);

    void pune(int i, int j) {
          if(i>=0 && i<marime) p[i] = j;
    }
    int da(int i) {
          return p[i];
    }
};

// constructor de copie
matrice::matrice(const matrice &a) {
    int i;

    p = new int[a.marime];
    if(!p) exit(1);
    for(i=0; i<a.marime; i++) p[i] = a.p[i];
}

main()
{
    matrice num(10);
    int i;

    for(i=0; i<10; i++) num.pune(i, i);
    for(i=9; i>=0; i--) cout << num.da(i);
    cout << "\n";

    // creeaza alta matrice si o initializeaza cu num
```



```
    matrice x=num; // apeleaza constructorul pentru copie
    for(i=0; i<10; i++) cout << x.da(i);

    return 0;
}
```

Când **num** este folosit pentru a iniţializa pe **x**, este apelat constructorul de copie, este alocată memoria pentru noua matrice şi este stocată în **x.p**, iar conţinutul din **num** este copiat în matricea lui **x**. În acest fel, **x** şi **num** conţin matrice care au aceleaşi valori, dar fiecare matrice este independentă şi distinctă. (Adică **num.p** şi **x.p** nu indică spre aceeaşi zonă de memorie.) Dacă funcţia constructor de copie nu ar fi fost creeată, iniţializarea implicită pe biţi ar fi determinat ca **x** şi **num** să utilizeze aceeaşi memorie pentru matricele lor. (Deci, **num.p** şi **x.p** ar fi indicat spre aceeaşi locaţie.)

Constructorul de copie este apelat doar pentru iniţializări. De exemplu, următoarea secvenţă nu apelează constructorul de copie definit în programul precedent.

```
matrice a(10);
    .
    .
    .
matrice b(10);
b = a; // nu apeleaza constructorul de copie
```

În acest caz, **b = a** efectuează operaţia de atribuire. Dacă = nu este supraîncărcat (şi aici nu este cazul), va fi efectuată o copie pe biţi. De aceea, în unele cazuri, pentru a evita problemele, veţi avea nevoie şi să supraîncărcaţi operatorul =, şi să creaţi un constructor pentru copie.

Iniţializare dinamică

Iniţializarea dinamică este procesul prin care variabilele sunt iniţializate în timpul rulării şi nu în timpul compilării. Mai mult, iniţializarea dinamică permite ca o variabilă să fie iniţializată folosind orice expresie validă în acel moment, inclusiv alte variabile şi apelări de funcţii. Atât C cât şi C++ permit iniţializarea dinamică a variabilelor locale. Dar, în C++, puteţi să le iniţializaţi dinamic şi pe cele globale. De exemplu, acest program este perfect valid:

```
#include <iostream.h>
#include <stdlib.h>

int i = atoi("1233"); // valida in C++, nu in C
```

```
int x = i * 2; // valida in C++, nu in C

main()
{
    cout << "valoarea lui i este " << i << endl;
    cout << "valoarea lui x este " << x << endl;
    return 0;
}
```

După cum era de aşteptat, programul afişează această ieşire:

```
valoarea lui i este 1233
valoarea lui x este 2466
```

Funcţii membre const şi volatile

Funcţiile membre ale claselor pot fi declarate ca **volatile**, **const** sau de ambele tipuri. Se aplică câteva reguli. Prima, obiectele declarate ca **volatile** pot apela doar funcţiile membre declarate, de asemenea, ca **volatile**. Un obiect de tip **const** nu poate apela o funcţie membru care nu este **const**. Dar, o funcţie membru **const** poate fi apelată atât de obiectele **const** cât şi de celelalte. O funcţie membru **const** nu poate modifica obiectul care o apelează. Pentru funcţiile care sunt atât **const** cât şi **volatile** regulile se combină.

Pentru a specifica o funcţie membru ca fiind **const** sau **volatile**, folosiţi formele prezentate în acest exemplu:

```
class X {
public:
    int f1() const; // functie membru const
    void f2(int a) volatile; // functie membru volatile
    char *f3() const volatile; // functie membru const volatile
};
```

Utilizarea cuvântului cheie asm

Puteţi să integraţi limbajul de asamblare direct în programul dvs. în C++ folosind cuvântul cheie **asm**. El are această sintaxă:

asm("*şir*");

Aici, *şir* este transmis, nemodificat, către asamblor.

Mai multe compilatoare admit trei forme generale, uşor diferite, ale instrucţiunii **asm**, ele fiind prezentate aici:

asm *instrucţiune*;
asm *instrucţiune* linie nouă
asm {
    secvenţă de instrucţiuni
}

Aici, *instrucţiune* este orice instrucţiune validă a limbajului de asamblare.

Ca un exemplu simplu (şi destul de „sigur"), acest program foloseşte **asm** pentru a executa o instrucţiune **INT 5**, care apelează funcţia print-screen:

```
//Afiseaza pe ecran.
#include <iostream.h>

main(void)
{
    asm int 5; // foloseste asm int 5
    return 0;
}
```

ATENŢIE: *Trebuie să posedaţi cunoştinţe temeinice în domeniul programării în limbajul de asamblare pentru a folosi instrucţiunea* ***asm****. Dacă nu aveţi experienţă în lucrul cu acest limbaj, este bine să evitaţi folosirea sa, deoarece pot rezulta erori foarte primejdioase.*

Specificaţii pentru editarea legăturilor

În C++ puteţi specifica modul de editare a legăturilor. De exemplu, puteţi să îi spuneţi compilatorului să editeze o funcţie ca fiind din C, din C++, sau, în funcţie de implementarea compilatorului de C++, ca fiind produsă de alt limbaj, cum ar fi FORTRAN. Implicit, funcţiilor li se editează legăturile ca fiind din C++. Dar, folosind *specificaţiile de editare a legăturilor*, puteţi să determinaţi ca unei funcţii să i se editeze legăturile ca aparţinând unui limbaj diferit. Forma generală a specificatorului de editare este:

extern "*limbaj*" *prototip-funcţie*

unde *limbaj* indică limbajul dorit. Toate compilatoarele de C++ vor admite editarea legăturilor pentru C şi pentru C++. Unele vor admite şi alte limbaje.

Următorul program determină ca **funcmeaC()** să aibă legăturile editate ca o funcție de C.

```
#include <iostream.h>

extern "C" void funcmeaC(void);

main(void)
{
    funcmeaC();
    return 0;
}

// Aceasta va avea legaturile editate ca o functie din C.
void funcmeaC(void)
{
    cout << "Acesteia i s-au editat legaturile ca pentru o
            functie din C.\n";
}
```

***NOTĂ:** Cuvântul cheie **extern** este o parte necesară pentru specificarea modului de editare de legături. Mai mult, specificarea trebuie să fie globală; ea nu poate fi folosită în interiorul unei funcții.*

Puteți specifica mai mult de o funcție o dată, folosind această formă de specificare:

extern "*limbaj*" {
prototipuri
}

Utilizarea unei specificări de editare de legături este rară și, probabil, nu veți avea nevoie să o folosiți.

Caracteristici noi adăugate de standardul ANSI C++ propus

În procesul de standardizare, comitetul ANSI C++ a adăugat mai multe caracteristici noi care nu au făcut parte din specificația originală de C++. Nu toate aceste adăugiri sunt implementate curent de orice compilator de C++ uzual. (Totuși, aceste noi facilități vor fi valabile, în viitorul apropiat, în practic toate

compilatoarele de C++.) Chiar dacă nici una dintre aceste noi facilități nu sunt, practic, indispensabile pentru folosirea pe deplin a limbajului C++, unele dintre ele vă oferă un control mai bun asupra anumitor situații. Altele sunt incluse pentru comoditate. În acest paragraf, vi se oferă o scurtă trecere în revistă a acestor caracteristici. Totuși, deoarece natura lor exactă este în curs de definire, pentru detalii privind implementarea lor va trebui să verificați manualul de utilizare a compilatorului dvs.

***NOTĂ:** Deoarece standardul de C++ este încă în stadiu de dezvoltare, nu există nici o garanție că vreuna dintre caracteristicile descrise în acest paragraf va fi definită de versiunea finală a acestuia. Dar este foarte probabil că toate aceste facilități vor face parte din standardul de C++.*

Noi operatori de modelare

Chiar dacă C++ admite în întregime operatorii clasici de modelare, standardul ANSI C++ propus definește încă patru. Ei sunt **const_cast**, **dynamic_cast**, **reinterpret_cast** și **static_cast**. Formele lor generale sunt:

const_cast<*tip*>(*obiect*)
dynamic_cast<*tip*>(*obiect*)
reinterpret_cast<*tip*>(*obiect*)
static_cast<*tip*>(*obiect*)

Aici, *tip* specifică tipul final al modelatorului, iar *obiect* este obiectul care este modelat în noul tip.

Operatorul **const_cast** este folosit pentru a înlătura atributele **const** și/sau **volatile**. Tipul final trebuie să fie același cu tipul sursă, cu excepția modificării caracteristicilor **const** și **volatile**. Cea mai uzuală întrebuințare pentru **const_cast** este eliminarea specificației **const**.

dynamic_cast efectuează o modelare în timpul rulării care verifică validitatea unei modelări. Dacă modelarea nu poate fi făcută, ea eșuează, iar expresia este evaluată ca null. Utilizarea sa principală este pentru modelări asupra tipurilor polimorfice. (Clasele polimorfice sunt clase care conțin funcții virtuale.) De exemplu, **dynamic_cast** poate să returneze un pointer către un obiect derivat, dându-se un pointer către o clasă de bază polimorfică. Dacă obiectul indicat nu este un obiect al clasei de bază sau al unei clase derivate, atunci **dynamic_cast** se evaluează ca null.

Operatorul **static_cast** efectuează o modelare nepolimorfică. De exemplu, el poate fi folosit pentru a modela un pointer al clasei de bază într-unul al unei clase derivate. El poate fi utilizat, de asemenea, pentru orice conversie standard. Operatorul **reinterpret_cast** modifică un tip într-unul fundamental diferit. De

exemplu, el poate fi folosit pentru a schimba un pointer într-un întreg. Un **reinterpret_cast** trebuie utilizat pentru modelarea tipurilor inerent incompatibile.

Doar **const_cast** poate să înlăture atributul **const**. Aceasta înseamnă că nici **dynamic_cast**, nici **static_cast** şi nici **reinterpret_cast** nu pot modifica o caracteristică de tip **const**.

Următorul program ilustrează utilizarea operatorului **reinterpret_cast**.

```
// Un exemplu care foloseste reinterpret_cast.
#include <iostream.h>

main()
{
    int i;
    char *p = "Acesta este un sir";

    i = reinterpret_cast<int> (p); // transforma un pointer
                                   // in intreg

    cout << i;

    return 0;
}
```

Tipul de date bool

Deşi, practic, nu este necesar, standardul ANSI C++ propus a adăugat tipul de date **bool**, care este capabil să păstreze o valoare booleană. Obiectele de tip **bool** pot avea doar valorile **true** (adevărat) sau **false** (fals). (**true** şi **false** sunt cuvinte-cheie care fac acum parte din limbajul C++.) Valorile de tip **bool** sunt ridicate automat la întregi, atunci când sunt folosite cu expresii ne-booleene.

Utilizarea unui nume pentru zona de influenţă

Standardul ANSI C++ propus a adăugat şi cuvântul cheie **namespace**, care poate fi folosit pentru a defini o sferă de acţiune. Iată, aici, forma generală pentru **namespace**:

```
namespace nume {
    // declarări de obiecte
}
```

De exemplu,


```
namespace NumeleZonei {
    int i, k;
    void funcmea(int j) { cout << j; }
}
```

Aici **i**, **k** şi **funcmea()** fac parte din sfera de acţiune definită de **NumeleZonei**.

Deoarece un nume al domeniului de influenţă defineşte o sferă de acţiune, pentru a vă referi la obiectele definite în interiorul său, va trebui să folosiţi operatorul de specificare a domeniului. De exemplu, pentru a atribui lui **i** valoarea 10, trebuie să folosiţi instrucţiunea:

```
NumeleSpatiu::i = 10;
```

Dacă membrii zonei de influenţă vor fi folosiţi frecvent, puteţi folosi o directivă **using** pentru a simplifica accesul la ei. Instrucţiunea **using** are două forme generale:

using namespace *nume*;
using *nume::membru*;

În prima formă, *nume* specifică numele pentru zona de influenţă la care doriţi acces. Toţi membrii care sunt definiţi în interiorul zonei indicate pot fi folosiţi fără specificator. În cea de-a doua formă, devine vizibil doar un obiect, acela menţionat explicit. De exemplu, presupunând **NumeZona** prezentat anterior, sunt valide următoarele instrucţiuni **using** şi atribuiri:

```
using NumeZona::k; // este vizibil doar k
k = 10; // corect, deoarece k este vizibil
using namespace NumeZona; // sunt vizibili toti membrii din
                          // NumeleZonei
i = 10; // OK deoarece acum sunt vizibili toti membrii din
        // NumeleZonei
```

*NOTĂ: Forma exactă şi natura operatorului **namespace** şi ale instrucţiunilor **using** sunt în curs de finalizare. Pentru detalii de implementare relativ la aceste două caracteristici, verificaţi manualul compilatorului dvs.*

Identificarea tipului în timpul rulării

Una dintre cele mai importante facilităţi adăugate de standardul ANSI C++ propus este *identificarea tipului în timpul rulării* (RTTI). Utilizând acest tip de identificare, puteţi să determinaţi tipul unui obiect în timpul executării programului. Pentru a se

obţine tipul unui obiect, se foloseşte **typeid**. Ca să îl puteţi folosi, trebuie să includeţi fişierul antet TYPEINFO.H. Forma sa generală este:

typeid(*obiect*)

obiect este aici obiectul al cărui tip doriţi să-l aflaţi. **typeid** returnează o referinţă către un obiect de tipul **type_info** care descrie tipul obiectului definit prin *obiect*. Clasa **type_info** defineşte următorii membri publici:

```
bool operator==(const type_info &ob) const;
bool operator!=(const type_info &ob) const;
bool before(const type_info &ob) const;
const char *name() const;
```

Compararea tipurilor este asigurată de == şi de != supraîncărcate. Funcţia **before()** returnează adevărat dacă obiectul care apelează se află înaintea obiectului folosit ca parametru, în ordinea citirii. (Această funcţie este, de obicei, doar pentru uz intern. Valoarea sa returnată nu are nici o legătură cu moştenirea sau cu ierarhizarea claselor.) Funcţia **name()** returnează un pointer către numele tipului. (Deoarece **typeid** este în curs de a fi definit, pentru detalii privind orice alte tipuri de operaţii definite de clasa **type_info**, va trebui să consultaţi manualul compilatorului dvs.)

Când **typeid** este aplicat unui pointer către clasa de bază al unei clase polimorfice, el va returna automat tipul obiectului către care indică, inclusiv numele oricărei clase derivate din acea bază. (Amintiţi-vă că o clasă polimorfică este una ce conţine cel puţin o funcţie virtuală.)

typeid este ilustrat de către următorul program:

```
// Un exemplu care foloseste typeid.
#include <iostream.h>
#include <typeinfo.h>

class ClasaBaza {
    int a, b;
    virtual void f() {}; // face ClasaBaza polimorfica
};

class Derivat1: public ClasaBaza {
    int i, j;
};

class Derivat2: public ClasaBaza {
    int k;
```



```
};

main()
{
    int i;
    ClasaBaza *p, obbaza;
    Derivat1 ob1;
    Derivat2 ob2;

    // Mai intai afiseaza numele tipului pentru un tip
    // incorporat.
    cout << "Tipul lui i este ";
    cout << typeid(i).name() << endl;

    // Prezinta typeid cu tipuri polimorfice.
    p = &obbaza;
    cout << "p indica spre un obiect de tipul ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "p indica spre un obiect de tipul ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "p indica spre un obiect de tipul ";
    cout << typeid(*p).name() << endl;

    return 0;
}
```

Iată ieşirea produsă de acest program:

```
Tipul lui i este int
p indica spre un obiect de tipul ClasaBaza
p indica spre un obiect de tipul Derivat1
p indica spre un obiect de tipul Derivat2
```

După cum s-a menţionat, când **typeid** este aplicat unui pointer al clasei de bază de tip polimorfic, tipul obiectului spre care indică va fi determinat în timpul rulării, după cum arată ieşirea produsă de către program.

Identificarea tipului în timpul rulării nu este ceva ce se foloseşte în orice program. Totuşi, când lucraţi cu tipuri polimorfice, ele vă permit să cunoaşteţi asupra cărui tip de obiect se operează într-o situaţie dată.

Constructori expliciţi

Standardul propus pentru ANSI C++ a definit cuvântul cheie **explicit**. El este folosit pentru a crea „constructori fără conversie". De exemplu, dându-se următoarea clasă,

```
class ClasaMea {
    int i;
public:
    ClasaMea(int j) {i = j;}
    // ...
};
```

obiectele din **ClasaMea** pot fi declarate astfel:

```
ClasaMea ob1(1);
ClasaMea ob2 = 10;
```

În acest caz, instrucţiunea

```
ClasaMea ob2 = 10;
```

este convertită automat în forma

```
ClasaMea ob2(10);
```

Dar, dacă funcţia constructor pentru **ClasaMea** este declarată **explicit**, această conversie automată nu va fi efectuată. Iată **ClasaMea** folosind un constructor **explicit**:

```
class ClasaMea {
    int i;
public:
    explicit ClasaMea(int j) {i = j;}
    // ...
};
```

Acum vor fi permişi doar constructori de forma:

```
ClasaMea ob(110);
```


Utilizarea operatorului mutable

Standardul ANSI C++ propus defineşte cuvântul cheie **mutable**. El este folosit pentru a permite ca un membru al unui obiect să suprascrie atributul **const**. Cu alte cuvinte, un membru **mutable** al unui obiect **const** nu este constant şi poate fi modificat.

Tipul wchar_t

Standardul ANSI propus pentru C++ defineşte tipul **wchar_t** care poate conţine caractere foarte lungi. De obicei ele sunt memorate pe 16 biţi şi sunt folosite pentru a reprezenta seturile de caractere ale limbajelor care au mai mult de 255 de caractere.

Fişiere antet noi

Propunerea pentru standardul ANSI C++ a definit o nouă cale de a introduce fişierele antet. Totuşi, stilul clasic (care este folosit de această carte) este admis pe deplin. Noul stil nu cere utilizarea efectivă a unui nume de fişier. În schimb, este folosit un specificator standard pentru antet, care, dacă este necesar, va fi transformat de către compilator într-un nume de fişier. De exemplu, noul stil al formatului de antet pentru includerea fişierelor sistemului de I/O este:

```
#include <iostream>
```

După cum puteţi vedea, grupul **.h** a fost lansat. Regula poate fi generalizată. De exemplu, folosind noul stil pentru formatul fişierului antet, următoarea instrucţiune include antetul pentru sistemul de I/O cu fişiere:

```
#include <fstream>
```

Va trebui să verificaţi în manualul compilatorului dacă acesta admite noul stil pentru specificarea fişierelor antet.

Diferenţe între C şi C++

În cea mai mare parte, C++ este un superstandard ANSI C şi, teoretic, toate programele în C sunt şi programe în C++. Totuşi, există câteva diferenţe, cele mai importante dintre ele fiind discutate aici.

În C++ variabilele locale pot fi declarate oriunde în interiorul unui bloc. În C, ele trebuie declarate la începutul blocului, înainte să apară orice instrucţiune de „acţiune".

Una dintre cele mai importante, deşi subtile, diferenţe dintre C şi C++ este că în C o funcţie declarată astfel

```
int f();
```

nu spune *nimic* despre nici un parametru al acesteia. Deci, atunci când între parantezele care urmează numelui funcţiei nu este specificat nimic, în C înseamnă că nu s-a afirmat nimic despre nici un parametru al acelei funcţii. Ea poate să aibă parametri, poate să nu aibă. Dar, în C++ o astfel de declaraţie de funcţie înseamnă că ea *nu* are parametri. Deci, în C++, aceste două declaraţii sunt echivalente:

```
int f();
int f(void);
```

În C++, **void** este opţional. Mulţi programatori de C++ includ **void** pentru a fi foarte clar pentru oricine citeşte un program în care o funcţie nu are nici un parametru, dar practic el nu este necesar.

În C++ toate funcţiile trebuie să aibă prototip. Acest lucru este opţional în C (chiar dacă o bună practică de programare recomandă folosirea deplină a prototipurilor într-un program în C).

O mică dar potenţial importantă diferenţă între C şi C++ este că în C, o constantă caracter este ridicată automat la rangul unui întreg. În C++ nu este aşa.

În C nu este o eroare să se declare o variabilă globală de mai multe ori, chiar dacă nu este o practică de programare prea bună. În C++ aceasta este o eroare.

În C, un identificator poate să aibă lungimea de maximum 31 de caractere. În C++ nu există astfel de limite. Totuşi, din punct de vedere practic, identificatorii extrem de lungi sunt incomozi şi rareori necesari.

În C, deşi nu se obişnuieşte, puteţi să apelaţi **main()** din interiorul programului dvs. Acest lucru nu este permis în C++.

În C nu puteţi să preluaţi adresa unei variabile de tip **register**. În C++ acest lucru este permis.

Partea a III-a

Câteva aplicaţii de C++

Partea a treia a acestei cărţi conţine câteva exemple de aplicaţii scrise în C++. Scopul acestei secţiuni este dublu. În primul rând, exemplele ajută la ilustrarea beneficiilor programării orientate pe obiecte, inclusiv avantajele polimorfismului, ale încapsulării şi moştenirii şi ale creării bibliotecilor de clase. Apoi, exemplele arată cum poate fi aplicat C++ pentru a rezolva diverse tipuri de probleme - orientate sau nu pe obiecte. Amintiţi-vă că C++ este o versiune îmbunătăţită şi lărgită de C, care oferă programatorului mai multă putere şi flexibilitate, independent de metodologia orientării pe obiecte. Deci, nu are neapărat importanţă dacă veţi folosi C++ pentru a realiza OOP sau doar pentru a vă ridica ştacheta peste nivelul normal al temelor dvs. de programare.

Capitolul 23

O clasă de tip şir

După cum ştiţi, şirurile sunt introduse în C++ ca matrice de caractere terminate cu null şi nu ca tipuri de date separate. Această abordare face ca şirurile din C++ să fie puternice, elegante şi eficiente. De asemenea, relaţia strânsă dintre matrice şi pointeri vă permite să scrieţi multe operaţii cu şiruri „scurte şi cu miez". Totuşi, de multe ori este necesar să folosiţi un şir, dar nu vă trebuie să stabiliţi un grad înalt de eficienţă şi de putere. În aceste cazuri, lucrul cu şiruri în C++ poate să devină o adevărată corvoadă. Din fericire, este posibil ca în C++ să creaţi un tip de şir care pierde oarecum din eficienţă, dar câştigă mult în uşurinţa utilizării.

În acest capitol este dezvoltată o clasă de tip şir, care face mult mai uşoară crearea, utilizarea şi manevrarea şirurilor.

NOTĂ: În momentul scrierii acestei cărţi, comitetul de standardizare ANSI C++ este în procesul de definire a unei clase standard de tip şir. (Dar, forma sa finală nu a fost încă stabilită.) Scopul acestui capitol nu este să prezinte o alternativă a acestei clase, ci de a vă oferi o imagine a uşurinţei în care orice tip nou de date poate fi adăugat şi integrat în mediul C++. Crearea unei clase de tip şir este un exemplu esenţial al acestui proces. Pe lângă faptul că noua clasă de tip şir din acest capitol este mult mai simplă decât cea din C++ standard, mai există un avantaj: ea vă oferă controlul complet al modului în care sunt implementate şi manevrate şirurile. O veţi găsi probabil folositoare în multe situaţii.

Definirea unui tip de şir

În primul rând este important să definim ce se înţelege printr-un tip de şir şi ce fel de operaţii pot fi efectuate cu el. Din fericire, alte limbaje au definit tipurile de şir şi pot fi luate ca modele pe care ne bazăm descrierea acestora. La prima vedere poate să pară ciudat, dar un model foarte bun pentru introducerea tipului de şiruri este BASIC. Chiar dacă majoritatea programatorilor de C++ nu sunt entuziasmaţi de acesta ca limbaj de programare în general, modul în care manevrează şiruri este intuitiv şi uşor de folosit. BASIC este, de asemenea, un model bun, deoarece îl cunosc practic toţi programatorii.

Clasa de tip şir prezentată în acest capitol nu copiază exact abordarea din BASIC, dar împrumută cele mai importante facilităţi, care vor fi studiate în continuare.

În BASIC, pentru a da o valoare unui şir, îi atribuiţi pur şi simplu un şir între ghilimele folosind operatorul de atribuire, =. De exemplu, aceasta este o atribuire validă în BASIC:

```
A$ = "Acesta este un sir"
```


(Toate variabilele de tip şir din BASIC trebuie să se termine cu un semn dolar. Desigur, clasa de tip şir prezentată în acest capitol nu va avea această restricţie.) Instrucţiunea de mai sus atribuie variabilei **A$** şirul **"acesta este un sir"**.

De asemenea, puteţi să atribuiţi o variabilă şir altei variabile şir. De exemplu, următoarea comandă copiază în **B$** şirul conţinut în **A$**.

```
B$ = A$
```

După cum puteţi vedea, diferenţa principală dintre BASIC şi C++ în ce priveşte atribuirea unui şir pentru o variabilă de tip şir este că BASIC foloseşte un operator, în timp ce C++ foloseşte o apelare a funcţiei **strcpy()** (deşi C++ permite ca matricele de tip caracter să fie *iniţializate* cu operatorul =).

Operatorul + este folosit în BASIC pentru a concatena două şiruri. De exemplu, această secvenţă face ca **C$** să conţină valoarea „Va salut".

```
A$ = "Va "
B$ = "salut"
C$ = A$ + B$
```

De fapt, secvenţa precedentă ar putea fi simplificată astfel:

```
A$ = "Va "
C$ = A$ + "salut"
```

Aici, o variabilă de tip şir este concatenată cu un şir cuprins între ghilimele. Astfel, BASIC permite unei variabile de tip şir să fie concatenată cu altă variabilă sau cu şiruri cuprinse între paranteze.

O diferenţă între concatenarea unui şir în BASIC şi funcţia standard **strcat()** este că aceasta din urmă modifică unul dintre şirurile apelate astfel încât să conţină rezultatul. În schimb, când alipiţi două şiruri în BASIC, apare un şir temporar care conţine concatenarea, iar şirurile originale rămân nemodificate.

Comparaţiile de şiruri în BASIC sunt intuitive deoarece folosesc aceiaşi operatori relaţionali folosiţi şi când se compară alte tipuri de date. Toate comparaţiile de şiruri sunt efectuate în ordine alfabetică. De exemplu, aceasta determină dacă **A$** este mai mare ca **B$**.

```
IF A$ > B$ THEN PRINT "A$ este mai mare ca B$"
```

După cum arată exemplul precedent, avantajul principal al abordărilor şirurilor C din BASIC este că el permite ca toate operaţiile importante să fie efectuate cu aceiaşi operatori folosiţi pentru alte tipuri de date. Într-un fel, BASIC supraîncarcă atribuirea, adunarea şi operatorii relaţionali astfel încât să poată lucra şi cu şiruri. Acesta este conceptul de bază care a fost introdus în acest capitol de către clasele

de tip şir. Tipul de şir introdus aici va înlocui apelările funcţiilor de bibliotecă cu operatori supraîncărcaţi.

Cunoscând abordarea şirurilor din BASIC, suntem acum gata să creăm o clasă de tip şir în C++.

Clasa StrType

Clasa de tip şir definită aici va îndeplini următoarele cerinţe:

- Şirurilor le pot fi atribuite valori cu operatorul de atribuire.
- Obiectelor de tip şir le pot fi atribuite alte obiecte de tip şir sau şiruri între ghilimele.
- Concatenarea a două obiecte de tip şir se realizează cu operatorul +.
- Operatorul - poate fi folosit pentru a exclude un subşir dintr-un şir.
- Comparările de şiruri sunt efectuate cu operatorii relaţionali.
- Un obiect de tip şir poate fi iniţializat ori folosind şiruri între ghilimele ori alte obiecte de tip şir.
- Şirurile trebuie să fie de lungimi arbitrare şi variabile. Aceasta implică pentru fiecare şir alocarea dinamică a memoriei.
- Va fi asigurată o metodă de convertire a obiectelor de tip şir în şiruri cu terminaţia null.

Clasa care va trata şirurile se numeşte **StrType**. Iată declararea ei:

```
class StrType {
    char *p;
    int marime;
public:
    StrType(char *sir);
    StrType();
    StrType(const StrType &o); // constructor de copie
    ~StrType() {delete [] p;}

    friend ostream &operator<<(ostream &stream, StrType &o);
    friend istream &operator>>(istream &stream, StrType &o);

    StrType operator=(StrType &O); // atribuie un obiect
                                   // tip StrType
    StrType operator=(char *s); // atribuie un sir intre
                                // ghilimele

    StrType operator+(StrType &o); /* concateneaza un obiect
                                      tip StrType */
    StrType operator+(char *s); /* concateneaza un sir intre
                                   ghilimele */
    friend StrType operator+(char *s, StrType &o); /* con-
             cateneaza un sir intre ghilimele cu un obiect
             tip StrType */

    StrType operator-(StrType &O); // exclud un subsir
    StrType operator-(char *s); /* exclud un subsir intre
                                   ghilimele */

    // operatii relationale intre obiecte de tip StrType
    int operator==(StrType &o); {return !strcmp(p, o.p);}
    int operator!=(StrType &o); {return strcmp(p, o.p);}
    int operator<(StrType &o); {return strcmp(p, o.p) < 0;}
    int operator>(StrType &o); {return strcmp(p, o.p) > 0;}
    int operator<=(StrType &o); {return strcmp(p, o.p) <= 0;}
    int operator>=(StrType &o); {return strcmp(p, o.p) >= 0;}

    // operatii intre obiecte de tip StrType si siruri
    // intre ghilimele
    int operator==(char *s) {return !strcmp(p, s);}
    int operator!=(char *s) {return strcmp(p, s);}
    int operator<(char *s) {return strcmp(p, s) < 0;}
    int operator>(char *s) {return strcmp(p, s) > 0;}
    int operator<=(char *s) {return strcmp(p, s) <= 0;}
    int operator>=(char *s) {return strcmp(p, s) >= 0;}

    int strsize() {return strlen(p);} /* returneaza
                                         lungimea sirului */

    void makestr(char *s) {strcpy(s, p);} // creeaza siruri
                                          // intre ghilimele

    operator char *() { return p;} // conversie in char *
};
```

Secţiunea particulară din **StrType** conţine doar două elemente: **p** şi **marime**. Când este creat un obiect de tip şir, memoria pentru păstrarea şirului este alocată dinamic de către **new**, iar în **p** este pus un pointer spre acea memorie. Şirul spre care indică **p** va fi unul normal, o matrice de tip caracter terminată cu un caracter de null. Chiar dacă nu este, practic, necesar, mărimea şirului este păstrată în **marime**. Deoarece şirul spre care indică **p** este unul normal, ar fi posibil să se

calculeze mărimea sa de câte ori este necesar. Totuşi, după cum veţi vedea, această valoare este folosită atât de des de către funcţiile membre din **StrType**, încât apelările repetate ale funcţiei **strlen()** nu sunt justificate.

Următoarele paragrafe detaliază cum lucrează clasa **StrType**.

Funcţiile constructor şi destructor

Un obiect de tip **StrType** poate fi declarat în trei feluri. El poate fi declarat fără nici o iniţializare ori iniţializat cu un şir între ghilimele sau cu un obiect de tip **StrType**. Constructorii care admit aceste trei operaţii sunt prezentaţi aici:

```
// Nici o initializare explicita.
StrType::StrType() {
   marime = 1; // face loc pentru terminatia de null
   p = new char[marime];
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }
   *p = '\0';
}

// Initializare folosind un sir intre ghilimele, terminat
// cu null.
StrType::StrType(char *sir) {
   marime = strlen(sir) + 1; // face loc pentru
                             // terminatorul null
   p = new char[size];
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }
   strcpy(p, sir);
}

// Initializare folosind un obiect StrType
StrType::StrType(const StrType&o) {
   marime = o.marime;
   p = new char[marime];
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }
   strcpy(p, o.p);
}
```


Când este creat un obiect tip **StrType** fără iniţializare, i se atribuie valoarea unui şir null. Deşi şirul s-ar putea să fi fost fi lăsat nedefinit, faptul că toate obiectele de tip **StrType** conţin un şir valid, terminat cu null, simplifică multe alte funcţii membre.

Când un obiect din clasa **StrType** este iniţializat printr-un şir între ghilimele, mai întâi se determină lungimea şirului. Valoarea sa este memorată în **marime**. Apoi, se alocă memoria necesară prin **new**. După ce verificarea de siguranţă confirmă că **p** nu este nul, şirul de iniţializare este copiat în zona de memorie spre care indică **p**.

Când este folosit un obiect de tip **StrType** pentru a iniţializa un altul, procesul este similar cu cel al utilizării unui şir între ghilimele. Singura diferenţă este aceea că mărimea unui şir este cunoscută şi nu mai trebuie calculată. O astfel de versiune a constructorului clasei **StrType** este şi constructorul de copii al clasei. Acesta va fi apelat ori de câte ori este folosit un obiect de tip **StrType** pentru a iniţializa un altul. Cu alte cuvinte, va fi apelat când sunt create obiecte temporare şi când obiectele de tip **StrType** sunt pasate spre funcţii. (Pentru constructorii de copii vedeţi **Capitolul 22**.)

Dându-se constructorii anteriori, sunt permise următoarele declaraţii:

```
StrType x("sirul meu"); // foloseste sir intre ghilimele
StrType y(x); // foloseste un alt obiect
StrType z; // nici o initializare explicita
```

Funcţia destructor pentru **StrType** eliberează pur şi simplu memoria spre care indică **p**.

I/O cu şiruri

Deoarece este foarte uzual să doriţi să introduceţi sau să obţineţi şiruri, clasa **StrType** supraîncarcă operatorii << şi >>, aşa cum se prezintă aici;

```
// Scrie un sir la iesire
ostream &operator<<(ostream &stream, StrType &o)
{
   stream << o.p;
   return stream;
}

// Introduce un sir.
istream &operator>>(istream &stream, StrType &o)
```

```
{
    char t[255]; // marime arbitrara - modificati-o daca
                 // este necesar
    int lung;

    for(lung=0; lung<255; lung++) {
          stream.da(t[lung]);
          if(t[lung])=='\n') break;
          if(t[lung]=='\b')
          if(lung) {
                lung--;
                cout << "'\b'";
          }
    }
    t[lung] = '\0';
    lung++;

    if(lung > o.marime) {
          delete o.p;
          o.p = new char[lung];
          if(!o.p) {
                cout << "Eroare de alocare\n";
                exit(1);
          }
          o.marime = lung;
    }
    strcpy(o.p, t);
    return stream;
}
```

După cum puteţi vedea, metoda este foarte simplă. Totuşi, reţineţi că parametrul **o** este transmis prin referinţă. Deoarece obiectele de tip **StrType** pot fi foarte mari, transmiterea lor prin referinţă este mai eficientă decât cea prin valoare. Din acest motiv, toţi parametrii din **StrType** sunt transmişi prin referinţă. (Orice funcţie pe care o creaţi şi care preia parametri din **StrType** va face, probabil, la fel.)

Introducerea şirurilor se dovedeşte a fi puţin mai dificilă decât obţinerea lor, deoarece nu se poate utiliza o instrucţiune ca următoarea pentru a se citi intrarea.

```
stream >> t;
```

Motivul este că operaţia de intrare normală care citeşte şirurile în stilul C++ se opreşte din citit când întâlneşte primul caracter de spaţiu liber. De aceea, este necesar să citiţi un şir introducând caracterele unul câte unul. Versiunea operatorului >> supraîncărcat pentru tipul **StrType** citeşte caractere până când se întâlneşte o linie nouă.

O dată ce şirul a fost citit, dacă mărimea noului şir depăşeşte pe cea păstrată curent în **o**, acea memorie este eliberată şi este alocată o cantitate mai mare. Apoi noul şir este copiat aici.

Funcţiile de atribuire

Puteţi să atribuiţi valori unui obiect de tip **StrType** în două feluri. În primul rând, puteţi să îi atribuiţi un alt obiect de tip **StrType**. În al doilea rând, puteţi să îi atribuiţi un şir între ghilimele. Iată aici cele două funcţii **operator=()** supraîncărcate care realizează aceste operaţii:

```
// Atribuie un obiect de tip StrType unui obiect de tip
// StrType.
StrType StrType::operator=(StrType &o)
{
    StrType temp(o.p);

    if(o.marime > marime) {
          delete p; // elibereaza memoria veche
          p = new char[o.marime];
          marime = o.marime;
          if(!p) {
                cout << "Eroare de alocare\n";
                exit(1);
          }
    }

    strcpy(p, o.p);
    strcpy(temp.p, o.p);

    return temp;
}

// Atribuie un sir intre ghilimele unui obiect de tip
// StrType.
StrType StrType::operator=(char *s)
{
    int lung = strlen(s) + 1;
    if(marime < lung) {
          delete p;
```

```
            p = new char[lung];
            marime = lung;
            if(!p) {
                cout << "Eroare de atribuire\n";
                exit(1);
            }
        }
        strcpy(p, s);
        return *this;
    }
```

Aceste două funcţii efectuează o primă verificare pentru a vedea dacă memoria spre care indică **p**, având ca ţintă obiectul tip **StrType**, este suficient de mare pentru a păstra ceea ce doriţi să copiaţi în ea. Dacă nu este aşa, memoria veche este eliberată şi se alocă una nouă. Apoi şirul este copiat în obiect şi se returnează rezultatul. Aceste funcţii permit următoarele tipuri de atribuire:

```
StrType x("test"), y;
y = x; // obiect StrType in obiect StrType
x = "un nou sir pentru x"; // sir intre ghilimele in obiect
                           // StrType
```

Fiecare funcţie de atribuire trebuie să returneze valoarea atribuită (valoarea din partea dreaptă), astfel încât să fie admise atribuiri multiple, ca aceasta:

```
StrType x, y, z;
x = y = z = "test";
```

Concatenarea

Concatenarea a două şiruri se realizează cu operatorul +. Clasa **StrType** permite următoarele trei situaţii de concatenare distincte:

- Concatenarea unui obiect de tip **StrType** cu un alt obiect **StrType**
- Concatenarea unui obiect de tip **StrType** cu un şir între ghilimele
- Concatenarea unui şir între ghilimele cu un obiect de tip **StrType**

Când este folosit în astfel de situaţii, operatorul + produce ca rezultate de ieşire un obiect de tip **StrType** care reprezintă concatenarea celor doi operanzi. El nu modifică nici unul dintre aceştia. (Această abordare diferă de funcţia **strcat()**, care modifică primul său argument.)

Iată prezentate funcţiile **operator+()** supraîncărcate:

```
// Concateneaza doua obiecte de tip StrType.
StrType StrType::operator+(StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;
    lung = strlen(o.p) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
        cout << "Eroare de alocare\n";
        exit(1);
    }
    strcpy(temp.p, p);

    strcat(temp.p, o.p);

    return temp;
}

// Concateneaza un obiect de tip StrType cu un sir intre
// ghilimele.
StrType StrType::operator+(char *s)
{
    int lung;
    StrType temp;

    delete temp.p;

    lung = strlen(s) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
        cout << "Eroare de alocare\n";
        exit(1);
    }
    strcpy(temp.p, p);

    strcat(temp.p, s);
```

```
    return temp;
}

// Concateneaza un sir intre ghilimele cu un obiect tip
// StrType.
StrType operator+(char *s, StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;

    lung = strlen(s) + strlen(o.p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(temp.p, s);

    strcat(temp.p, o.p);

    return temp;
}
```

Toate trei funcţiile lucrează, în esenţă, în acelaşi fel. Mai întâi este creat un obiect de tip **StrType** numit **temp**. Acest obiect va conţine rezultatul unei concatenări şi este obiectul returnat de către funcţie. Apoi este eliberată memoria spre care indică **temp.p**. Motivul acestei operaţii este că, atunci când este creat **temp**, se alocă doar un octet de memorie (pentru a reţine un loc), deoarece nu există o iniţializare explicită. Apoi, se alocă suficientă memorie pentru a păstra concatenarea celor două şiruri. În final, cele două şiruri sunt copiate în memoria spre care indică **temp.p** şi este returnat **temp**.

Excluderi de subşiruri

O funcţie utilă pentru şiruri, care nu se găseşte în multe limbaje, este cea de excludere a subşirurilor. Aşa cum este introdusă în clasa **StrType**, funcţia de *excludere de subşiruri* elimină toate apariţiile unui subşir specificat dintr-un alt şir. Ea este realizată cu operatorul -.

Clasa **StrType** admite două cazuri de excludere de subşiruri. Unul permite ca un obiect de tip **StrType** să fie exclus din alt obiect de acelaşi tip. Celălalt face

posibilă eliminarea unui şir între ghilimele dintr-un obiect de tip **StrType**. În următorul exemplu sunt prezentate cele două funcţii **operator-()**:

```
// Exclude un subsir dintr-un sir folosind obiecte de tip
// StrType.
StrType StrType::operator-(StrType &subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;

    s1 = p;
    for(i=0; *s1; i++) {
          if(*s1!=*subsir.p) { // daca nu este prima litera
                               // a subsirului
                temp.p[i] = *s1; // atunci copiaza in temp
                s1++;
          }
          else { // poate fi extras
                for(j=o; subsir.p[j]==s1[j] && subsir.p[j];
                    j++);
                if(!subsir.p[j]) { // este subsirul, deci
                                   // trebuie eliminat
                      s1 += j;
                      i--;
                }
                else { // nu este subsirul, continua copierea
                      temp.p[i] = *s1;
                      s1++;
                }
          }
    }
    temp.p[i] = '\0';
    return temp;
}

// Exclude siruri intre ghilimele din obiecte de tip
// StrType.
StrType StrType::operator-(char *subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;
```

```
        s1 = p;
        for(i=0; *s1; i++) {
              if(*s1!=*subsir.p) { // daca nu este prima litera
                                   // din subsir
                    temp.p[i] = *s1; // atunci copiaza in temp
                    s1++;
              }
              else {
                    for(j=o; subsir.p[j]==s1[j] && subsir.p[j];
                         j++);
                    if(!subsir.p[j]) { // este subsirul, deci
                                       // trebuie eliminat
                          s1 += j;
                          i--;
                    }
                    else { // nu este subsirul, continua copierea
                          temp.p[i] = *s1;
                          s1++;
                    }
              }
        }
        temp.p[i] = '\0';
        return temp;
}
```

Aceste funcţii lucrează prin copierea în **temp** a conţinutului operandului stâng, eliminând în timpul procesului orice apariţie a subşirului specificat de operandul drept. Este returnat obiectul rezultat de tip **StrType**. Se înţelege că, în timpul procesului, nu este modificat nici un operand.

Clasa **StrType** permite excluderi de subşiruri ca acestea:

```
StrType x("Imi place C++"), y("place");
StrType z;

z = x - y; // z va contine "Imi C++"
z = x - "C++"; // z va contine "Imi place"

// sunt eliminate aparitii multiple
z = "ABCDABCD";
x = z - "A"; // contine "BCDBCD"
```


Operatorii relaţionali

Clasa **StrType** admite ca şirurilor să le fie aplicată întreaga gamă a operatorilor relaţionali. Operatorii relaţionali sunt redefiniţi (overload) în interiorul declarării clasei **StrType**. Pentru claritate ei sunt repetaţi aici:

```
// operatii relationale intre obiecte de tip StrType
int operator==(StrType &o)  {return !strcmp(p, o.p);}
int operator!=(StrType &o)  {return strcmp(p, o.p);}
int operator<(StrType &o)  {return strcmp(p, o.p) < 0;}
int operator>(StrType &o)  {return strcmp(p, o.p) > 0;}
int operator<=(StrType &o)  {return strcmp(p, o.p) <= 0;}
int operator>=(StrType &o)  {return strcmp(p, o.p) >= 0;}

// operatii intre obiecte de tip StrType si siruri intre
// ghilimele
int operator==(char *s) {return !strcmp(p, s);}
int operator!=(char *s) {return strcmp(p, s);}
int operator<(char *s) {return strcmp(p, s) < 0;}
int operator>(char *s) {return strcmp(p, s) > 0;}
int operator<=(char *s) {return strcmp(p, s) <= 0;}
int operator>=(char *s) {return strcmp(p, s) >= 0;}
```

Operaţiile relaţionale sunt foarte intuitive; nu ar trebui să aveţi probleme cu înţelegerea definiţiilor lor. Totuşi, reţineţi că **StrType** introduce doar comparaţia între două obiecte de acest tip sau între un obiect de tip **StrType** ca operand stâng şi un şir între ghilimele ca operand drept. Dacă vreţi să puteţi scrie şirul aflat între ghilimele în partea stângă, iar obiectul de tip **StrType** în partea dreaptă, va trebui să mai adăugaţi funcţii relaţionale.

Dându-se funcţiile supraîncărcate pentru operatori relaţionali redefinite de **StrType**, sunt permise următoarele tipuri de comparare a şirurilor:

```
StrType x("unu"), y("doi"), z("trei");

if(x < y) cout << "x mai mic decit y";

if(z=="trei") cout << "z egal trei";
y = "u";
z = "nu";
if(x==(y+z)) cout << "x este egal cu y+z";
```

Diverse funcţii pentru şiruri

Clasa **StrType** defineşte trei funcţii care fac ca obiectele de tip **StrType** să se integreze şi mai bine în rândul şirurilor obişnuite din C++. Ele sunt **strsize()**, **makestr()** şi funcţia de conversie **operator char*()** şi sunt definite în interiorul declaraţiei clasei **StrType.** Iată, mai jos, aceste funcţii:

```
int strsize() {return strlen(p);} // returneaza marimea
                                   // sirului
void makestr(char *s) {strcpy(s, p);} // formeaza siruri
                                       // intre ghilimele
operator char *() {return p;} // converteste in sir
```

Primele două funcţii sunt uşor de înţeles. După cum puteţi vedea, funcţia **strsize()** returnează lungimea unui şir spre care indică **p**. Funcţia **makestr()** copiază şirul spre care indică **p** într-o matrice de caractere. Această funcţie este folositoare când doriţi să obţineţi un şir terminat cu null, pornind de la un obiect de tip **StrType**.

Funcţia de conversie **operator char*()** returnează **p**, care este, desigur, un pointer către şirul conţinut de obiect. Această funcţie permite ca un obiect de tip **StrType** să fie folosit oriunde poate fi folosit şi un şir terminat cu null. De exemplu, acesta este un cod corect:

```
StrType x("Hello");
// afiseaza sirul folosind o functie C++ standard
puts(x); // conversie automata in char *
```

Amintiţi-vă că o funcţie de conversie este executată automat atunci când un obiect este implicat într-o expresie pentru care este definită acea conversie. În acest caz, deoarece prototipul funcţiei **puts()** cere compilatorului un argument de tip **char ***, se efectuează automat conversia din **StrType** în **char**, producând returnarea unui pointer către şirul conţinut în **x**. Datorită funcţiei de conversie, puteţi să folosiţi un obiect de tip **StrType** în locul unui şir normal, între ghilimele, ca argument pentru orice funcţie care poate prelua un argument de tip **char ***.

NOTĂ: *Conversia în* ***char **** *încalcă principiul încapsulării, deoarece o dată ce o funcţie are un pointer spre şirul obiectului, este capabilă să modifice direct acel şir, evitând funcţiile membre din* ***StrType*** *şi fără ca să fie implicat obiectul. De aceea, trebuie să utilizaţi cu multă grijă conversia în* ***char ****. *(Dacă nu aveţi nevoie de ea, mai bine eliminaţi-o pur şi simplu dintre specificaţiile clasice. Pierderea încapsulării în acest caz este compensată de utilitatea şi integrarea sporită în cadrul funcţiilor de bibliotecă existente. Schimbul însă nu este întotdeauna avantajos.*

Întreaga clasă StrType

Iată listingul întregii clase **StrType** împreună cu o scurtă funcţie **main()** care îi ilustrează facilităţile:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>

class StrType {
    char *p;
    int marime;
public:
    StrType(char *sir);
    StrType();
    StrType(const StrType &0); // constructor de copie
    ~StrType() {delete [] p;}

    friend ostream &operator<<(ostream &stream, StrType &o);
    friend istream &operator>>(istream &stream, StrType &o);

    StrType operator=(StrType &o); // atribuie un obiect de
                                   // tip StrType
    StrType operator=(char *s); // atribuie un sir intre
                                // ghilimele

    StrType operator+(StrType &o); // concateneaza un
                                   // obiect de tip StrType
    StrType operator+(char *s); // concateneaza un sir
                                // intre ghilimele
    friend StrType operator+(char *s, StrType &o); /* con-
         cateneaza un sir intre ghilimele cu un obiect de
         tip StrType */

    StrType operator-(StrType &o); // exclude un subsir
    StrType operator-(char *s); // exclude un subsir intre
                                // ghilimele

    // operatii relationale intre obiecte de tip StrType
    int operator==(StrType &o) {return !strcmp(p, o.p);}
    int operator!=(StrType &o) {return strcmp(p, o.p);}
```

```
    int operator<(StrType &o) {return strcmp(p, o.p) < 0;}

    int operator>(StrType &o) {return strcmp(p, o.p) > 0;}
    int operator<=(StrType &o) {return strcmp(p, o.p) <= 0;}
    int operator>=(StrType &o) {return strcmp(p, o.p) >= 0;}

    // operatii intre obiecte de tip StrType si siruri
    // intre ghilimele
    int operator==(char *s) {return !strcmp(p, s);}
    int operator!=(char *s) {return strcmp(p, s);}
    int operator<(char *s) {return strcmp(p, s) < 0;}
    int operator>(char *s) {return strcmp(p, s) > 0;}
    int operator<=(char *s) {return strcmp(p, s) <= 0;}
    int operator>=(char *s) {return strcmp(p, s) >= 0;}

    int strsize() {return strlen(p);} // returneaza
                                      // lungimea sirului
    void makestr(char *s) {strcpy(s, p);} // creeaza siruri
                                          // intre ghilimele
    operator char *() { return p;} // conversie in char *
};

// Nici o initializare explicita
StrType::StrType() {
    marime = 1; // face loc pentru terminatia de null
    p = new char[marime];
    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(p, " ");
}

// Initializare folosind un sir intre ghilimele
StrType::StrType(char *str) {
    marime = strlen(sir) + 1; // face loc pentru terminatia
                              // de null
    p = new char[marime];
    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(p, sir);
```



```
}

// Initializare folosind un obiect de tip StrType.
StrType::StrType(const StrType &o) {
    marime = o.marime;
    p = new char[marime];
    if(!p) {
          cout << "Eroare de alocare\n;
          exit(1);
    }
    strcpy(p, o.p);
}

// Scrie un sir la iesire
ostream &operator<<(ostream &stream, StrType &o)
{
    stream << o.p;
    return stream;
}

// Introduce un sir.
istream &operator>>(istream &stream, StrType &o)
{
    char t[255]; // marime arbitrara - modificati-o daca
                 // este necesar
    int lung;

    for(lung=0; lung<255; lung++) {
         stream.da(t[lung]);
         if(t[lung]=='\n') break;
         if(t[lung]=='\b')
               if(lung) {
                     lung--;
               cout << "'\b'";
         }
    }
    t[lung] = '\0';
    lung++;

    if(lung > o.marime) {
         delete o.p;
         o.p = new char[lung];
         if(!o.p) {
```

```
                cout << "Eroare de alocare\n";
                exit(1);
          }
          o.marime = lung;
    }
    strcpy(o.p, t);
    return stream;
}

// Atribuie un obiect de tip StrType unui obiect de tip
// StrType.
StrType StrType::operator=(StrType &o)
{
    StrType temp(o.p);
    if(o.marime > marime) {
          delete p; // elibereaza memoria veche
          p = new char[o.marime];
          marime = o.marime;
          if(!p) {
                cout << "Eroare de alocare\n";
                exit(1);
          }
    }

    strcpy(p. o.p);
    strcpy(temp.p, o.p);

    return temp;
}
// Atribuie un sir intre ghilimele unui obiect de tip
// StrType.
StrType StrType::operator=(char *s)
{
    int lung = strlen(s) + 1;
    if(marime < lung) {
          delete p;
          p = new char[lung];
          marime = lung;
          if(!p) {
                cout << "Eroare de alocare\n";
                exit(1);
          }
    }
```



```
    strcpy(p, s);
    return *this;
}

// Concateneaza doua obiecte de tip StrType.
StrType StrType::operator + (StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;
    lung = strlen(o.p) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(temp.p, p);

    strcat(temp.p, o.p);

    return temp;
}

// Concateneaza un obiect de tip StrType cu un sir intre
// ghilimele.
StrType StrType::operator+(char *s)
{
    int lung;
    StrType temp;

    delete temp.p

    lung = strlen(s) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(temp.p, p);
```

```
    strcat(temp.p, s);

    return temp;
}

// Concateneaza un sir intre ghilimele cu un obiect de tip
// StrType.
StrType operator+(char *s, StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;

    lung = strlen(s) + strlen(o.p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    strcpy(temp.p, s);
    strcat(temp.p, o.p);
    return temp;
}

// Exclude un subsir dintr-un sir folosind obiecte de tip
// StrType.
StrType StrType::operator-(StrType &subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;

    s1 = p;
    for(i=o; *s1; i++) {
          if(*s1!=*subsir.p) { // daca nu este prima litera
                               // a subsirului
                temp.p[i] = *s1; // atunci copiaza in temp
                s++;
          }
          else {
                for(j=0; subsir.p[j]==s1[j] && subsir.p[j];
                    j++);
```



```
                if(!subsir.p[j]) { // este subsirul, deci
                                   // trebuie eliminat
                      s1 += j;
                      i--;
                }
                else { // nu este subsirul, continua copierea
                      temp.p[i] = *s1;
                      s1++;
                }
          }
    }
    temp.p[i] = '\0';
    return temp;
}

// Exclude siruri intre ghilimele din obiecte de tip
// StrType.
StrType StrType::operator-(char *subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;

    s1 = p;
    for(i=0; *s1; i++) {
          if(*s1!=*subsir) { // daca nu este prima litera
                             // din subsir
                temp.p[i] = *s1; // atunci copiaza in temp
                s1++;
          }
          else {
                for(j=0; subsir[j]==s1[j] && subsir[j];
                    j++);
                if(!subsir[j]) { // este subsirul, deci
                                 // trebuie eliminat
                      s1 += j++;
                      i--;
                }
                else { // nu este subsirul, continua copierea
                      temp.p[i] = *s1;
                      s1++;
                }
          }
```

```
    }
    temp.p[i] = '\0';
    return temp;
}

main()
{
    StrType s1("O sesiune-exemplu folosind obiecte de tip
                sir.\n");
    StrType s2(s1);
    StrType s3;
    char s[80];

    cout << s1 << s2;

    s3 = s1;
    cout << s1;

    s3.makestr(s);

    cout << "Converteste intr-un sir: " << s;

    s2 = "Acesta este un sir nou.";
    cout << s2 << endl;

    StrType s4(" Asa este.");
    s1 = s2+s4;
    cout << s1 << endl;

    if(s2==s3) cout << "Sirurile sunt egale.\n";
    if(s2!=s3) cout << "Sirurile nu sunt egale.\n";
    if(s1<s4) cout << "s1 mai mic decat s4\n";
    if(s1>s4) cout << "s1 mai mare decat s4\n";
    if(s1<=s4) cout << "s1 mai mic sau egal cu s4\n";
    if(s1>=s4) cout << "s1 mai mare sau egal cu s4\n";

    if(s2 > "ABC") cout << "s2 mai mare decat ABC\n\n";

    s1 = "unu doi trei unu doi trei\n";
    s2 = "doi";
    cout << "Sirul initial: " << s1;
    cout << "Sirul dupa excluderea lui doi: ";
    s3 = s1 - s2;
```



```
    cout << s3;

    cout << endl;
    s4 = "Va salut!";
    s3 = s4 + " sirurile din C++ sunt nostime\n";
    cout << s3;
    s3 = s3 - "Va salut!";
    s3 = "Nu-i asa ca sunt" + s3;
    cout << s3;

    s1 = s3 - "sunt ";
    cout << s1;
    s3 = s1;

    cout << "Introduceti un sir: ";
    cin >> s1;
    cout << s1 << endl;
    cout << "s1 are lungimea de " << s1.strsize()
         << " caractere.\n";

    puts(s1); // converteste in char *

    s1 = s2 = s3;
    cout << s1 << s2 << s3;

    s1 = s2 = s3 = "Pa ";
    cout << s1 << s2 << s3;

    return 0;
}
```

Precedentul program determină următoarea ieşire:

```
O sesiune-exemplu folosind obiecte de tip sir.
O sesiune-exemplu folosind obiecte de tip sir.
O sesiune-exemplu folosind obiecte de tip sir.
Converteste intr-un sir: O sesiune-exemplu folosind obiecte de tip sir.
Acesta este un sir nou.
Acesta este un sir nou. Asa este.
Sirurile nu sunt egale.
s1 este mai mare decat s4
s1 este mai mare sau egal ca s4
s2 este mai mare decat ABC
```

```
Sirul initial: unu doi trei unu doi trei
Sirul dupa excluderea lui doi: unu trei unu trei

Va salut! sirurile din C++ sunt nostime
Nu-i asa ca sunt sirurile din C++ sunt nostime
Nu-i asa ca sirurile din C++ nostime
Introduceti un sir: Imi place C++
si are lungimea de 13 caractere.
Imi place C++
Nu-i asa ca sirurile din C++ nostime
Nu-i asa ca sirurile din C++ nostime
Nu-i asa ca sirurile din C++ nostime
Pa Pa Pa
```

Această ieşire presupune că şirul "Imi place C++" a fost scris de către utilizator la solicitarea unei introduceri.

Pentru a avea acces uşor la clasa **StrType**, îndepărtaţi funcţia **main()** şi puneţi restul listingului precedent într-un fişier numit STR.H. Apoi intoduceţi pur şi simplu acest fişier antet în orice program în care doriţi să folosiţi **StrType**.

Utilizarea clasei StrType

Drept concluzii ale acestui capitol, sunt oferite două exemple care ilustrează cum poate fi folosită clasa **StrType**. Primul exemplu creează un simplu dicţionar care foloseşte obiecte de tip **StrType**. El construieşte la început o matrice bidimensională de obiecte de acest tip. În fiecare pereche de şiruri, primul conţine cuvântul de bază după care se poate face căutarea, iar al doilea o listă de sinonime sau cuvinte înrudite. Se solicită un cuvânt, iar dacă acesta se află în dicţionar, sunt afişate sinonimele. Program este foarte simplu, dar reţineţi modul elegant în care sunt tratate şirurile, care se datorează utilizării clasei **StrType** şi operatorilor săi. (Amintiţi-vă că fişierul antet STR.H conţine clasa **StrType**.)

```
#include "str.h"
#include <iostream.h>

StrType dictionar[][2] = {
    "carte", "volum, tom",
    "magazin", "pravalie, shop, butic",
    "pistol", "arma, revolver, pistolet",
    "alergare", "fuga, jogging, cros",
    "gindire", "meditare, contemplare, reflectie",
    "calcul", "socoteala, estimare, rezolvare",
```



```
    "", ""
};

main()
{
    StrType x;

    cout << "Introduceti un cuvant: ";
    cin >> x;

    int i;
    for(i=0; dictionar[i][0]!=""; i++)
         if(dictionar[i][0]==x) cout << dictionar[i][1];
    return 0;
}
```

Următorul exemplu foloseşte un obiect de tip **StrType** pentru a afla dacă există o versiune executabilă a unui fişier pornind de la numele acestuia. Pentru a folosi programul, specificaţi în linia de comandă numele fişierului, fără nici o extensie. Programul va căuta în mod repetat un fişier executabil cu acel nume, la care adaugă o extensie, va încerca să-l deschidă şi va raporta rezultatul. (Dacă fişierul nu există, nu poate fi deschis.) După încercarea unei extensii, aceasta este exclusă din numele fişierului şi se adaugă una nouă. Din nou clasa **StrType** şi operatorii ei fac ca manevrarea şirurilor să fie uşoară şi lesne de urmărit.

```
#include "str.h"
#include <iostream.h>
#include <fstream.h>

// extensii pentru fisiere executabile
char ext[3][4] = {
    "EXE",
    "COM",
    "BAT"
};

main(int argc, char *argv[])
{
    StrType numef;
    int i;
    if(argc!=2) {
          cout << "Utilizare: numef\n";
          return 1;
```

```
    }

    numef = argv[1];
    numef = numef + "."; // adauga un punct
    for(i=0; i<3; i++) {
        numef = numef + ext[i]; // adauga extensia
        cout << "Incearca " << numef << " ";
        ifstream f(numef);
        if(f) {
            cout << "- Exista\n";
            f.close();
        }
        else cout << "- Nu exista\n";
        numef = numef - ext[i]; // exclude extensia
    }
    return 0;
}
```

De exemplu, dacă programul de mai sus se numeşte ESTEEXEC şi presupunând că există TEST.EXE, linia de comandă **ESTEEXEC TEST** determină această ieşire:

```
Incearca TEST.EXE - Exista
Incearca TEST.BAT - Nu exista
Incearca TEST.COM - Nu exista
```

Un lucru de reţinut despre program este acela că este folosit un obiect de tip **StrType** la apelarea funcţiei **fopen()**. Operaţia funcţionează deoarece este apelată automat funcţia de conversie **operator char*()**. După cum ilustrează această situaţie, prin aplicarea atentă a facilităţilor de C++, puteţi să realizaţi o integrare semnificativă a tipurilor create de dvs. în cadrul celor standard din C++.

Capitolul 24

O clasă pentru afişarea ferestrelor

Fereastra care se deschide pe ecran a devenit una dintre cele mai cunoscute imagini în lumea calculatoarelor moderne. Este greu de închipuit astăzi scrierea unei aplicaţii comerciale care să nu folosească una sau mai multe ferestre. Modelul ferestrelor este universal. Fereastra care se deschide reprezintă, de asemenea, un exemplu clasic de programare orientată pe obiecte folosind C++. De aceea, acest capitol dezvoltă o clasă de ferestre bazate pe text şi asigură toate funcţiile necesare unui sistem de ferestre simplu, dar eficient.

De vreme ce Microsoft Windows a devenit sistemul de operare preponderent pentru PC, puteţi să vă întrebaţi dacă mai are valoare practică crearea altui sistem de ferestre. Răspunsul este da, din trei motive. Primul, sistemul creat în acest capitol se bazează pe text (nu pe grafică, aşa ca în Windows). El este proiectat special pentru a rula sub DOS sau sub emulatorul de DOS asigurat de către Windows. Este mic şi eficient şi permite multă uşurinţă în lucrul cu ferestre bazate pe text. Al doilea motiv pentru a vă crea propriul sistem de ferestre este acela că puteţi controla complet sistemul pe care îl creaţi. Deoarece echipamentul hard este în continuă dezvoltare, puteţi să vă dezvoltaţi sistemul de ferestre pentru a beneficia de avantajul unor noi moduri video sau echipamente, înainte ca facilităţile pentru acestea să devină accesibile. Puteţi, de asemenea, să îl faceţi să lucreze în noi medii şi sisteme de operare. În sfârşit, sistemul de ferestre dezvoltat în acest capitol are interfaţă directă cu echipamentul hard video pentru PC. Ca atare, el ilustrează cum C++ este capabil să intre în legătură la un nivel scăzut cu dispozitivele, cât şi cu programele de nivel înalt. Aceleaşi metode generale, care creează interfaţa pentru echipamentul video pot fi folosite şi pentru alte tipuri de dispozitive.

NOTĂ: Sistemele de ferestre sunt, în mare măsură, dependente de echipamente. Acest capitol presupune un mediu PC, iar multe funcţii operează direct cu ROM BIOS (sistemul de bază de I/O) sau chiar cu hardul. Dar, modificând doar câteva funcţii, puteţi să adaptaţi sistemul de ferestre la orice mediu.

Înainte de a începe, este important să definim ce va face un sistem de ferestre.

Ferestrele

O fereastră este o porţiune din ecran folosită pentru un anumit scop. La deschiderea ei, se salvează ceea ce se află pe ecran în mod curent şi, în locul său, se afişează fereastra. Când se încheie aplicaţia ce foloseşte fereastra, aceasta este eliminată şi este restaurat conţinutul iniţial al ecranului. Este posibil ca pe ecran să existe, în acelaşi timp, mai multe ferestre.

O caracteristică importantă a sistemului de ferestre este aceea că el nu trebuie să permită unei aplicaţii ce foloseşte fereastra să scrie peste limitele acesteia. Deoarece mărimea ferestrei nu este neapărat cunoscută aplicaţiei, rămâne în

sarcina rutinei lor pentru ferestre, şi nu a aplicaţiei, să prevină suprascrierea. De aceea, nu pot fi folosite nici rutinele normale din C++ de I/O la consolă (cum ar fi **printf()** şi **gets()**), şi nici streamurile **cin** şi **cout**; ele trebuie înlocuite cu funcţii alternative de I/O specifice ferestrelor. De fapt, aceste funcţii formează o mare parte a oricărui sistem de ferestre.

Pentru a înţelege cum pot fi folosite efectiv ferestrele, imaginaţi-vă că aţi scris un editor de texte care conţine câteva facilităţi suplimentare. Una dintre ele este un calculator de buzunar. Deoarece utilizarea sa nu face, practic, parte din editorul de texte, are sens să folosim o fereastră oricând dorim să-l folosim. Astfel, când se lucrează cu acest calculator, editarea textului este doar suspendată, nu este complet întreruptă. După efectuarea calculelor, fereastra calculatorului este închisă şi continuă editarea textului.

Crearea unor funcţii de suport video

Înainte de a construi un sistem de ferestre, este necesar să dezvoltaţi câteva funcţii de suport. Deoarece crearea ferestrelor necesită controlul direct şi eficient al ecranului, sunt necesare funcţii specializate pentru a efectua operaţii de I/O cu acesta. După cum s-a mai spus, nu este posibil să folosiţi funcţiile şi operatorii normali din C++ pentru ieşiri. Funcţiile specializate vor ocoli atât DOS cât şi BIOS şi vor scrie direct la echipamentul hardware video. Acesta este singurul mod prin care se poate realiza reactualizarea rapidă a ecranului.

Pentru început, să aruncăm o privire asupra sistemului video al calculatorului.

NOTĂ: Discuţia prezentată în acest capitol despre componenta video a calculatorului este suficientă pentru a înţelege cum lucrează sistemul de ferestre. Pentru a învăţa mai mult despre interfaţa video în general, consultaţi şi alte cărţi.

Sistemul video al calculatorului

Toate calculatoarele conţin un adaptor oarecare care produce imagini pe monitor. Cele mai întâlnite patru tipuri de adaptoare sunt adaptoarele monocrome, adaptoarele grafice/color (CGA), adaptoarele grafice îmbunătăţite (EGA) şi matricea pentru grafică video (VGA). (Mai există şi o serie de adaptoare, Super VGA, care acceptă moduri video extinse, dar, în cadrul acestui capitol, VGA şi Super VGA se vor considera echivalente.) CGA, EGA şi VGA au mai multe moduri de operare. Sistemul de ferestre dezvoltat în acest capitol cere ca sistemul video să fie în mod text cu 80 de coloane, modul implicit de operare al majorităţii aplicaţiilor bazate pe consolă ce au scopuri generale. La adaptorul monocrom, modul text cu 80 de coloane are numărul 7. La adaptoarele CGA/EGA/VGA se foloseşte modul 2 sau 3.

Caracterele afişate pe ecran sunt păstrate într-o zonă în RAM, rezervată pentru adaptoarelor de afişare. Locaţia memoriei video RAM începe la B800:0000H. Deşi CGA, EGA şi VGA funcţionează diferit în unele moduri, ele operează identic în modurile 2 şi 3.

Fiecare caracter afişat pe ecran necesită doi octeţi de memorie video. Primul octet păstrează caracterul efectiv, iar cel de-al doilea păstrează *atributul său pentru ecran*. Pentru adaptoarele color, octetul de atribut este interpretat aşa cum se arată în Tabelul 24-1. Culorile primare pot fi combinate pentru a produce culori derivate. Dacă posedaţi un CGA, EGA sau VGA, caracterele sunt afişate implicit cu octetul atribut având valoarea 7. Aceasta determină activarea pentru text a celor trei culori, care produc culoarea albă. Pentru a produce negativul video, biţii din prim-plan sunt dezactivaţi iar cei trei de fundal sunt activaţi, obţinându-se valoarea 70H.

Adaptorul monocrom recunoaşte octeţii de strălucire şi de intensitate. Din fericire, el este proiectat pentru a interpreta atributul 7 ca normal, iar 70H ca negativ video. În plus, valoarea 1 determină sublinierea caracterelor.

Fiecare adaptor are de fapt de patru ori mai multă memorie decât îi este necesară pentru a afişa un text în modul cu 80 de coloane. Motivele pentru aceasta sunt două. În primul rând, memoria suplimentară este necesară pentru grafică (desigur, cu excepţia adaptoarelor monocrome). În al doilea rând, ea permite păstrarea în RAM a mai multor ecrane pe care le schimbă, atunci când este necesar. Fiecare regiune de memorie este numită *pagină video*, iar efectul schimbării paginii video curente este de-a dreptul spectaculos. Implicit DOS şi Windows folosesc pagina 0 şi, practic, toate aplicaţiile procedează la fel. Din acest motiv, ea va fi utilizată şi în rutinele acestui capitol. Dar, puteţi, dacă doriţi, să folosiţi alte pagini.

Există trei căi de a avea acces la adaptorul video. Prima este prin apelări către sistemul de operare, o metodă care este mult prea lentă pentru ferestre. A doua este prin rutine BIOS, metodă ceva mai rapidă, ce asigură suficientă viteză la

Bit	Valoare	Semnificaţie la activare
0	1	Prim-plan albastru
1	2	Prim-plan verde
2	4	Prim-plan roşu
3	8	Intensitate mică
4	16	Fundal albastru
5	32	Fundal verde
6	64	Fundal roşu
7	128	Caracter ce clipeşte

Tabelul 24-1 Octetul pentru atributul video

calculatoare puternice în cazul unor ferestre mici. Al treilea mod este prin citire şi scriere direct în RAM, cale care este foarte rapidă, dar necesită efort de programare mare. Totuşi, pentru ca ferestrele să apară „instantaneu" pe ecran este necesar accesul direct la memoria RAM video. Din acest motiv, cea de-a treia cale va fi folosită pentru sistemul de ferestre dezvoltat aici.

Pentru a permite ca sistemul de ferestre să aibă acces la memoria RAM video, este necesar un pointer către ea. Însă, când rulaţi un program în sistemul DOS (sau în emulatorul DOS din Windows), memoria RAM video se află într-un segment diferit de cel folosit de program. Aceasta înseamnă că trebuie utilizat un pointer FAR. Pointerii de tip FAR pot fi recunoscuţi de către un compilator de C++ într-unul din două feluri. Primul, prin cuvântul cheie suplimentar **far**, admis de multe compilatoare. (Unele compilatoare numesc acest cuvânt **_far** sau **__far**, verificaţi deci în manualul compilatorului dvs.) El permite ca un pointer să fie declarat ca FAR. A doua cale este să compilaţi programul folosind un model de memorie mare, caz în care implicit toţi pointerii sunt FAR. Rutinele folosite în acest capitol utilizează modelatorul de tip **far**. Dacă doriţi, puteţi să îl eliminaţi şi să compilaţi simplu codul folosind un model de memorie mare.

Accesul la BIOS

Chiar dacă funcţiile video care citesc şi scriu efectiv informaţii vor ocoli DOS şi BIOS şi vor avea acces direct la RAM, BIOS va mai fi folosit pentru câteva operaţii. Apelările la acesta vor fi efectuate folosind o întrerupere software. Funcţia care generează aşa ceva este **int86()**. Ea este specifică mediilor DOS şi are următorul prototip:

int int86(int *numar*, union REGS **registruintrare*, union REGS **registruiesire*);

BIOS foloseşte mai multe întreruperi diferite pentru diverse scopuri şi discutarea lor depăşeşte scopul acestui capitol. Oricum, cea legată de adaptorul video este întreruperea 16 (10H). Ca multe întreruperi pentru BIOS, întreruperea 16 are mai multe opţiuni, care sunt selectate în funcţie de valoarea registrului AH. Dacă funcţia BIOS întoarce o valoare, ea este, în general, returnată în AX. Totuşi, câteodată sunt folosite alte registre, dacă se întorc mai multe valori. Valoarea returnată de **int86()** este valoarea registrului **AX**.

Structura **REGS** este inclusă în fişierul antet DOS.H. Ea este definită astfel:

```
struct BYTEREGS {
    unsigned char al;
    unsigned char ah;
    unsigned char bl;
    unsigned char bh;
    unsigned char cl;
```

```
    unsigned char ch;
    unsigned char dl;
    unsigned char dh;
};

struct WORDREGS {
    unsigned ax;
    unsigned bx;
    unsigned cx;
    unsigned dx;
    unsigned si;
    unsigned di;
    unsigned cflag;
    unsigned flags; // Acesta nu este definit de toate
                   // versiunile de WORDREGS
};

union REGS {
    struct BYTEREGS h;
    struct WORDREGS x;
};
```

După cum puteţi vedea, **REGS** este o uniune de două structuri. Folosind structura **WORDREGS**, ea vă permite accesul la regiştrii unităţii centrale de prelucrare (CPU) ca recipiente de 16 biţi. **BYTREREGS** vă oferă acces la regiştrii individualizaţi de 8 biţi.

*NOTĂ: Microsoft C++ apelează funcţia **int86()**prin **_int86()** şi se referă la **REGS** ca **_REGS**.*

Determinarea locaţiei memoriei RAM video

Când citiţi şi scrieţi direct în memoria RAM video, mai întâi aveţi nevoie să depăşiţi problema produsă de faptul că adaptorul monocolor are memoria RAM video la B000:0000H, în timp ce celelalte îl au la B800:0000H. Pentru ca rutinele ferestrelor să opereze corect pentru fiecare adaptor, trebuie să ştie ce adaptor se află în sistem. Din fericire, există o cale simplă de a determina aceasta. Întreruperea BIOS 16, funcţia 15, returnează modul video curent. Cum s-a spus mai devreme, rutinele din acest capitol necesită modurile 2, 3 sau 7. Modurile 2 şi 3 pot fi utilizate doar de către CGA, EGA sau VGA, iar acestea nu pot folosi modul 7 - doar adaptorul monocolor o poate face. De aceea, dacă modul video curent este 7, înseamnă că este folosit un adaptor monocolor; altfel, el este de tipul CGA, EGA

sau VGA. Deoarece în modul text acestea din urmă funcţionează identic, pentru sistemul de ferestre nu contează cu care dintre ele se lucrează. Astfel, folosind modul video curent, este posibil să activaţi un pointer tip FAR global spre adresa memoria RAM video. Acest lucru este efectuat de către următoarele funcţii:

```
char far *vid_mem; // pointer spre memoria ecran in modul text

void set_v_ptr()
{
    int vmod;
    vmod = mod_video();
    if((vmod!=2) && (vmod!=3) && (vmod!=7)) {
        cout << "Video trebuie sa fie in modul text cu 80
                de coloane.";
        exit(1);
    }
    // stabileste adresa corecta a memoriei RAM video
    if(vmod==7) vid_mem = (char far *) 0xB0000000;
    else vid_mem = (char far *) 0xB8000000;
}

// Returneaza modul video curent.
mod_video()
{
    union REGS r;

    r.h.ah = 15; // da modul video
    return int86(0x10, &r, &r) & 255;
}
```

Funcţia **mod_video()** foloseşte întreruperea BIOS 16, funcţia 15, pentru a obţine modul video curent. Această valoare este folosită, apoi, pentru a determina spre ce va indica **vid_mem**, declarat ca un **char far *** global (către memoria folosită fie de adaptorul monocolor, fie de către alte adaptoare). Acest pointer va fi apoi utilizat de către rutinele bazate pe ferestre care cer acces la memoria RAM video.

Scrierea în memoria RAM video

O dată ce a fost determinat modul video şi a fost obţinut un pointer către memoria RAM video, pot fi create două funcţii de ieşire directe către aceasta. Aceste funcţii nu fac parte propriu-zis din sistemul de ferestre, dar ele sunt necesare deoarece

asigură o ieşire foarte rapidă. Funcţiile scriu un caracter sau un şir în locaţia X,Y. Este completat, de asemenea, octetul de atribut. Ele sunt prezentate aici:

```
// Scrie caracter cu atribut specificat.
void scrie_car(int x, int y, char ch, int atrib)
{
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2;
    *v++ = ch; // scrie caracterul
    *v = atrib; // scrie atributul
}

// Afiseaza un sir cu atributul specificat.
void scrie_sir(int x, int y, char *p, int atrib)
{
    register int i;
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2; // calculeaza adresa
    for(i=y; *p; i++) {
        *v++ = *p++; // scrie caracterul
        *v = atrib; // scrie atributul
    }
}
```

Deoarece în modul text cu 80 de coloane fiecare linie a ecranului are 80 de caractere, sunt folosiţi pentru fiecare linie 160 de octeţi (80 de octeţi pentru caractere şi 80 pentru atribute). Adresa pentru locaţia corectă în memoria RAM video este deci 160 ori coordonata Y plus de două ori coordonata X.

Poziţionarea cursorului

Când sunt efectuate operaţii de I/O folosind direct I/O din memoria RAM video, locaţia cursorului nu este actualizată automat. Aceasta înseamnă că rutinele pentru ferestre trebuie să deplaseze explicit cursorul. Următoarea funcţie efectuează această operaţie. (Funcţia nu face parte efectiv din clasa de ferestre, dar este utilizată de către aceasta. Funcţia foloseşte întreruperera BIOS 16, funcţia 2.)

```
// Trimite cursorul in pozitia X, Y specificata.
void goto_xy(int x, int y)
```



```
{
    union REGS r;

    r.h.ah = 2; // functia de adresare catre cursor
    r.h.dl = x; // coordonata coloanei
    r.h.dh = y; // coordonata rindului
    r.h.bh = 0; // pagina video
    int86(0x10, &r, &r);
}
```

Multe compilatoare asigură o funcţie „goto xy". Dacă al dvs. o are, sunteţi liber să o folosiţi în locul acesteia.

Clasa fereastră

Acum, deoarece a fost stabilită baza, poate fi dezvoltat sistemul de ferestre. Acesta este administrat de către clasa **wintype**. Iată declararea sa:

```
class wintype {
    // stabileste unde se plaseaza fereastra pe ecran
    int stgx; // coordonatele din stanga sus
    int susy;
    int drptx; // coordonatele din dreapta jos
    int josy;

    int chenar; // daca nu este 0, se afiseaza chenarul
    int activ; // nu este zero daca fereastra este afisata
               // pe ecran
    char *titlu; // mesaj pentru titlu

    int cursx, cursy; // localizarea curenta a cursorului
                      // in fereastra

    char *buf; // indica spre memoria buffer a ferestrei
    char color; // culoarea textului

    // functii particulare
    void salv_ecran(); // salveaza ecranul, astfel incat sa
                       // poata fi restaurat
    void restaur_ecran(); // restaureaza ecranul original
    void trasat_chenar(); // traseaza chenarul ferestrei
    void afisat_titlu(); // afiseaza titlul
public:
```

```
        wintype(int lx, int uy, // sus stanga
              int rx, int ly, // jos dreapta
              int b = 1, // diferit de zero pentru chenar
              char *mesaj = "" // mesaj pentru titlu
        );
  ~wintype() {winremove(); delete [] buf;}

  void winput(); // afiseaza o fereastra
  void winremove(); // sterge o fereastra
  int winputs(char *s); // scrie un sir in fereastra
  int winxy(int x, int y); // se deplaseaza la X, Y
                           // relativ la fereastra
  void wingets(char *s); // introduce un sir dintr-o
                         // fereastra
  int wingetche(); // introduce un caracter dintr-o
                   // fereastra
  void wincls(); // sterge continutul ferestrei
  void wincleol(); // sterge pana la sfarsitul liniei

  void setcolor(char c) {color = c;}
  char getcolor() {return color;}
  void setbkcolor(char c) {color = color | (c<<4);}
  char getbkcolor() {return (color>>4) & 127;}

  friend wintype &operator<<(wintype &o, char *s);
  friend wintype &operator>>(wintype &o, char *s);
};
```

Poziţia ferestrei pe ecran şi mărimea sa sunt determinate de variabilele **stgx**, **susy**, **drptx** şi **josy**. Acestea păstrează coordonatele colţurilor din stânga sus şi dreapta jos ale ferestrei.

Dacă fereastra va avea un chenar, atunci **chenar** trebuie să fie diferit de zero. Ori de câte ori fereastra se află pe ecran, **activ** este diferit de zero. Când fereastra nu este afişată, **activ** este zero. Titlul ferestrei (dacă există unul) este indicat de către **titlu**.

Localizarea curentă a cursorului în cadrul ferestrei este memorată în **cursx** şi **cursy**. Toate ieşirile către fereastră sunt poziţionate relativ la aceasta. Aceasta înseamnă că **cursx** şi **cursy** sunt relative la fereastră, nu la ecran. Deci, dacă **cursx** este 5 iar **cursy** este 3, atunci, indiferent unde se află fereastra în cadrul ecranului, cursorul este localizat la 5, 3 *în cadrul ferestrei*. Colţul din stânga sus al ferestrei este 0, 0.

Când este afişată o fereastră, conţinutul curent al ecranului este salvat în memoria indicată de **buf**. (Această memorie este alocată dinamic atunci când este

creată fereastra.) Culoarea textului este determinată de către valoarea lui **color**. Ea trebuie să fie una dintre valorile enumerate mai jos:

```
/* Culorile textului, primele 7 pot fi folosite, de
   asemenea, pentru a specifica culoarea fundalului. */
const enum clr {negru, albastru, verde, cian, rosu,
                magenta, maron, gri_deschis, gri_inchis,
                albastru_deschis, verde_deschis,
                cian_deschis, rosu_deschis,
                magenta_deschis, galben, alb,
                clipitor=128};
```

Iată funcţia constructor pentru **wintype**:

```
// Construieste o fereastra.
wintype::wintype(int lx, int uy, // sus stinga
         int rx, int ly, // jos dreapta
         int b, // diferit de zero pentru chenar
         char *mesaj // mesaj pentru titlu
    )
{
    if(lx<0) lx = 0;
    if(rx>79) rx = 79;
    if(uy<0) uy = 0;
    if(ly>24) ly = 24;

    stgx = lx; susy = uy;
    drptx = rx, josy = ly;
    chenar = b;
    titlu = mesaj;
    activ = 0;
    cursx = cursy = 0;
    buf = new char[2*(drptx-stgx+1)*(josy-susy+1)];
    if(!buf) {
        cout << "Eroare de alocare.\n";
        exit(1);
    }
    color = alb;
}
```

Funcţia constructor are grijă, pentru început, ca cele patru valori ale coordonatelor să fie în domeniu. Apoi iniţializează datele particulare. Ea alocă, de asemenea, memoria care va fi folosită pentru a salva conţinutul ecranului atunci

când se deschide fereastra. Când aceasta este dezactivată, va fi restaurat conținutul anterior al ecranului.

Rețineți că, implicit, culoarea textului este albă, cursorul este localizat la 0,0 iar fereastra nu este afișată.

*NOTĂ: Constructorul **wintype()** doar construiește o fereastră. El nu o și afișează. Afișarea unei ferestre este o operație separată de construirea ei.*

Destructorul **~wintype()** distruge fereastra (dacă este necesar) și apoi eliberează memoria spre care indică **buf**.

Afișarea și ștergerea unei ferestre

O dată construită o fereastră, ea poate fi afișată printr-o apelare a funcției **winput()**. Iată această funcție:

```
// Afiseaza o fereastra.
void wintype::winput()
{
    // activeaza fereastra
    if(!activ) { // nu este folosita in acel moment
          salv_ecran(; // salveaza ecranul curent
          activ = 1;
    }
    else return; // deja pe ecran

    if(chenar) trasat_chenar();
    afisat_titlu();

    // pozitioneaza cursorul in coltul din stanga sus
    goto_xy(stgx + cursx + 1, susy + cursy + 1);
}
```

Această funcție verifică la început dacă fereastra este deja pe ecran. În caz afirmativ, apelarea funcției **winput()** este ignorată. Altfel, conținutul curent al zonei de ecran în care va fi afișată fereastra va fi salvat prin apelarea funcției **salv_ecran()**, iar **activ** capătă valoarea 1. Dacă variabila **chenar** nu este zero, ferestrei i se desenează un chenar. Apoi este afișat titlul. În sfârșit, cursorul este plasat în fereastră. Deoarece valorile pentru **cursx** și **cursy** sunt relative la fereastră, ele trebuie să fie adunate cu **stgx** și respectiv cu **susy**, astfel încât cursorul să poată fi afișat în poziția corectă relativ la ecran.

În continuare este prezentată funcția **salv_ecran()**, funcție particulară din **wintype()**. Ea copiază pur și simplu, octet cu octet, conținutul din memoria RAM video referitoare la zona în care va fi afișată fereastra în memoria spre care indică **buf**, după care șterge acea zonă a ecranului.


```
// Salveaza ecranul, el putand fi restaurat dupa
// indepartarea ferestrei.
void wintype::salv_ecran()
{
    register int i, j;
    char *buf_ptr;
    char far *v, far *t;

    buf_ptr = buf;
    v = vid_mem;
    for(i=susy; i<josy+1; i++) {
          for(j=stgx; j<drptx+1; j++) {
                t = (v + (i*160) + j*2);
                *buf_ptr++ = *t++;
                *buf_ptr++ = *t;
                *(t-1) = ' '; sterge fereastra
          }
    }
```

În continuare sunt prezentate funcțiile **trasat_chenar()** și **afisat_titlu()**, de asemenea, funcții particulare din **wintype**.

```
// Traseaza un chenar in jurul ferestrei.
void wintype::trasat_chenar()
{
    register int i;
    char far *v, far *t;

    v = vid_mem;
    t = v;
    for(i=stgx+1; i<drptx; i++) {
          v += (susy*160) + i*2;
          *v++ = 196;
          *v = color;
          v = t;
          v += (josy*160) + i*2;
          *v++ = 196;
          *v = color;
          v = t;
    }
```

```
    for(i=susy+1; i<josy; i++) {
          v += (i*160) + stgx*2;
          *v++ = 179;
          *v = color;
          v = t;
          v += (i*160) + dreptx*2;
          *v++ = 179;
          *v = color;
          v = t;
    }
    // traseaza colturile
    scrie_car(stgx, susy, 218, color);
    scrie_car(stgx, josy, 192, color);
    scrie_car(drptx, susy, 191, color);
    scrie_car(drptx, josy, 217, color);
}

// Afiseaza titlul ferestrei.
void wintype::afisat_titlu()
{
    register int x, lung;

    x = stgx;

    /* Calculeaza pozitia de pornire corecta pentru a centra
       titlul - daca este negativ, mesajul nu se va incadra.
    */
    lung = strlen(titlu);
    lung = (drptx - x - lung) / 2;
    if(lung<0) return; // nu il afiseaza
    x = x + lung + 1;
    scrie_sir(x, susy, titlu, color);
}
```

Valorile 196 şi 179 din setul de caractere extins al calculatorului corespund liniilor orizontale şi verticale. Valorile folosite la sfârşitul funcţiei **trasat_chenar()** sunt caracterele pentru colţuri. Titlul este afişat doar dacă are loc în fereastră.

Pentru a îndepărta o fereastră de pe ecran, folosiţi funcţia **winremove()**, prezentată în continuare. Ea copiază înapoi în memoria RAM video ceea ce este păstrat în memoria spre care indică **buf**.

```
// Indeparteaza fereastra si restaureaza continutul
// initial al ecranului.
```



```
void wintype::winremove()
{
    if(!activ) return; // nu poate sa indeparteze o
                       // fereastra inactiva
    restaur_ecran(); // restaureaza ecranul initial
    activ = 0; // restaurare video
}
```

În afară de a da variabilei **activ valoarea** 0, scopul principal al funcţiei este de a restaura conţinutul iniţial al ecranului prin apelarea funcţiei particulare **restaur_ecran()**, aşa cum se arată mai jos:

```
// Restaureaza o zona a ecranului.
void wintype::restaur_ecran()
{
    register int i, j;
    char far *v, far *t;
    char *buf_ptr;

    buf_ptr = buf;
    v = vid_mem;
    t = v;
    for(i=susy; i<josy+1; i++)
        for(j=stgx; j<drptx+1; j++) {
            v = t;
            v += (i*160) + j*2;
            *v++ = *buf_ptr++; // scrie caracterul
            *v = *buf_ptr++; // scrie atributul
    }
}
```

I/O pentru ferestre

Intrarea din şi ieşirea într-o fereastră pot fi efectuate ori cu funcţiile membre ori cu operatorii supraîncărcaţi << şi >>. Toate operaţiile de I/O pentru ferestre trebuie să fie efectuate prin rutine care să împiedice depăşirea limitelor acestora.

Funcţia de intrare de nivel scăzut se numeşte **wingetche()**. Ea citeşte un caracter de la tastatură şi îl scrie în ecou în fereastră, aşa cum se arată aici:

```
/* Introduce o tasta apasata in interiorul unei ferestre.
   Returneaza codul complet pe 16 biti al tastei.
*/
```

```
int wintype::wingetche()
{
    union inkey {
          char ch[2];
          int i;
    } c;
    union REGS r;
    if(!activ) return 0; // fereastra nu este activa

    winxy(cursx, cursy);

    r.h.ah = 0; // citeste o tasta
    c.i = int86(0x16, &r, &r);

    if(c.ch[0]) {
          switch(c.ch[0]) {
                case '\r': // este apasata tasta ENTER
                      break;
                case '\b': // back space
                      break;
                default:
                      if(cursx + stgx < drptx - 1) {
                            scrie_char(stgx+ cursx + 1,
                            susy + cursy + 1, c.ch[0], color);
                            cursx++;
                      }
          }
          if(cursy < 0) cursy = 0;
          if(cursy + susy > josy - 2)
                cursy--;
          winxy(cursx, cursy);
    }
    return c.i;
}
```

Această funcţie apelează întreruperea BIOS 16, funcţia 0, care aşteaptă o apăsare de tastă şi returnează codul complet pe 16 biţi al tastei. Acesta este împărţit în două: caracterul şi codul de poziţie. Dacă tasta apăsată este o tastă caracter, acesta este returnat în cei 8 biţi de ordin inferior. Dacă însă este apăsată o tastă specială, pentru care nu există cod de caracter, cum ar fi o săgeată, atunci octetul de ordin inferior este zero, iar cel de ordin superior conţine codul de poziţie al tastei. De exemplu, codul de poziţie al săgeţilor în sus şi în jos este 72 şi, respectiv, 80. Deşi nici o funcţie pentru ferestre nu utilizează acest cod, el este prevăzut aici deoarece foarte probabil aplicaţiile dvs. cu ferestre vor avea nevoie de acces atât la caractere cât şi la coduri de poziţie.

După cum puteţi vedea examinând funcţia, nici o tastare nu va avea ecou dincolo de limitele ferestrei. De asemenea, caracterele se afişează în culoarea definită curent pentru fereastră. În sfârşit, fereastra trebuie să fie activă.

Pentru a citi un şir de la tastatură, folosiţi **wingets()**, prezentat aici:

```
// Citeste un sir dintr-o fereastra.
void wintype::wingets(char *s)
{
   char ch, *temp;

   temp = s;
   for(;;) {
    ch = wingetche();
    switch(ch) {
      case '\r': // este apasata tasta ENTER
         *s = '\0';
         return;
      case '\b': // backspace
         if(s>temp) {
              s--;
              cursx--;
              if(cursx < 0) cursx = 0;
              winxy(cursx, cursy);
              scrie_char(stg x + cursx + 1, susy + cursy +
                     1, ' ', color);
         }
         break;
      default: *s = ch;
         s++;

    }
}
```

Aceasta funcţie apelează **wingetche()** pentru a introduce fiecare caracter. Deoarece **wingetche()** împiedică afişările să treacă de limitele unei ferestre, acestea nu pot fi depăşite nici de vreo intrare din **wingets()**.

Pentru a obţine la ieşire un şir într-o fereastră, folosiţi **winputs()**, prezentată aici:

```
/* Scrie un sir in pozitia curenta a cursorului
   in fereastra specificata.
```

```
   Returneaza 0 daca fereastra nu este activa;
   altfel, returneaza 1.
*/
int wintype::winputs(char *s)
{
   register int x, y;
   char far *v;

   // se asigura ca fereastra este activa
   if(!activ) return 0;
   x = cursx + stgx + 1;
   y = cursy + susy + 1;

   v = vid_mem;
   v += (y*160) + x*2; // calculeaza adresa de pornire

   for( ; *s; s++) {
         if(y >= josy) {
               return 1;
         }
         if(x >= drptx) {
               return 1;
         }

         if(*s=='\n') {
               y++;
               x = stgx + 1;
               v = vid_mem;
               v += (y*160) + x*2; // calculeaza adresa
               cursy++; // incrementeaza Y
               cursx = 0; // aduce pe x la 0;
         }
         else {
               cursx++;
               x++;
               *v++ = *s; // scrie caracterul
               *v++ = color; // culoare
         }
         winxy(cursx, cursy);
   }
   return 1;
}
```


Această funcţie afişează şirul specificat începând de la poziţionarea cursorului (dată de **cursx** şi de **cursy**), în culoarea curentă. Nu va fi permisă nici o ieşire dincolo de limitele ferestrei.

După cum s-a menţionat, în afară de funcţiile membre de I/O, puteţi, de asemenea, să afişaţi un şir folosind << şi să introduceţi unul cu >>. În continuare sunt prezentaţi aceşti operatori supraîncărcaţi:

```
// Iesire intr-o fereastra.
wintype &operator<<(wintype &o, char *s)
{
   o.winputs(s);
   return o;
}
// Intrare de la o fereastra.
wintype &operator>>(wintype &o, char *s)
{
   o.wingets(s);
   return o;
}
```

Dacă doriţi să introduceţi alt tip de date, supraîncărcaţi din nou **<<** şi **>>**.

Alte trei funcţii de ieşire pentru ferestre sunt **wincls()**, care şterge conţinutul ferestrei, **wincleol()**, care şterge de la poziţia curentă a cursorului până la sfârşitul liniei, şi **winxy()**, care plasează cursorul în poziţia X, Y relativă la fereastră. Iată aceste funcţii:

```
// Sterge continutul unei ferestre.
void wintype::wincls()
{
   register int i, j;
   char far *v, far *t;

   v = vid_mem;
   t = v;
   for(i=susy+1; i<josy; i++)
         for(j=stgx+1; j<drptx; j++ {
               v = t;
               v += (i*160) + j*2;
               *v++ = ' '; // scrie un spatiu
               *v = color; // in culoarea fundalului
         }
   cursx = 0;
   cursy = 0;
```

```
}

// Sterge pana la sfarsitul liniei.
void wintype::wincleol()
{
    register int i, x, y;

    x = cursx;
    y = cursy;
    winxy(cursx, cursy);

    for(i=cursx; i<drptx-1; i++)
          winputs(" ");
    winxy(x, y);
}

/* Plaseaza cursorul intr-o fereastra intr-o pozitie
   specificata. Returneaza 0 daca este in afara limitelor;
   altfel, diferit de zero.
*/
int wintype::winxy(int x, int y)
{
    if(x<0 || x+stgx >= right-1)
          return 0;
    if(y<0 || y+susy >= josy-1)
          return 0;
    cursx = x;
    cursy = y;
    goto_xy(leftx+x+1, susy+y+1);
    return 1;
}
```

Puteţi să stabiliţi culoarea de prim-plan folosind **setcolor()**, iar cea de fundal cu **setbkcolor()**. Aceste funcţii sunt definite în interiorul declarării clasei **wintype**. Pentru **setcolor()** puteţi folosi orice culoare specificată în enumerarea **clr**. Pentru **setbkcolor()** puteţi să utilizaţi primele şapte culori. Funcţiile **getbkcolor()** şi **getcolor()** returnează culorile pentru fundal şi, respectiv, pentru prim-plan. Ele sunt definite inline în interiorul clasei **wintype**.

Întregul sistem de ferestre

Iată întregul sistem de ferestre, plus funcţiile de suport şi o funcţie **main()** care ilustrează funcţiile pentru ferestre:


```
// O clasa de ferestre.

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <bios.h>

// Functii globale
int mod_video();
void goto_xy(int x, int y);
void set_v_ptr();
void scrie_car(int x, int y, char ch, int attrib);
void scrie_sir(int x, int y, char *p, int attrib);

/* Culorile textului, primele 7 pot fi folosite, de
   asemenea, pentru a specifica culoarea fundalului. */
const enum clr {negru, albastru, verde, cian, rosu,
                magenta, maron, gri_deschis, gri_inchis,
                albastru_deschis, verde_deschis,
                ciano_deschis, rosu_deschis,
                magenta_deschis, galben, alb,
                clipitor=128};

char far *vid_mem; // pointer spre memoria ecranului in
                   // modul text

class wintype {
   // stabileste unde se plaseaza fereastra pe ecran
   int stgx; // coordonatele din stanga sus
   int susy;
   int drptx; // coordonatele din dreapta jos
   int josy;

   int chenar; // daca nu este 0, se afiseaza chenarul
   int activ; // nu este zero daca fereastra este afisata
              // pe ecran
   char *titlu; // mesaj pentru titlu

   int cursx, cursy; // localizarea curenta a cursorului
                     // in fereastra
```

```
    char *buf; // indica spre memoria buffer a ferestrei

    char color; // culoarea textului

    // functii particulare
    void salv_ecran(); // salveaza ecranul, astfel incat sa
                       // poata fi restaurat
    void restaur_ecran(); // restaureaza ecranul original
    void trasat_chenar(); // traseaza chenarul ferestrei
    void afisat_titlu(); // afiseaza titlul
    public:
          wintype(int lx, int uy, // sus stanga
                int rx, int ly, // jos dreapta
                int b = 1, // diferit de zero pentru chenar
                char *mesaj = "" // mesaj pentru titlu
    );

    ~wintype() {winremove(); delete [] buf;}

    void winput(); // afiseaza o fereastra
    void winremove(); // sterge o fereastra
    int winputs(char *s); // scrie un sir in fereastra
    int winxy(int x, int y); // se deplaseaza la X, Y
                             // relativ la fereastra
    void wingets(char *s); // introduce un sir dintr-o
                           // fereastra
    int wingetche(); // introduce un caracter dintr-o
                     // fereastra
    void wincls(); // sterge continutul ferestrei
    void wincleol(); sterge pana la sfarsitul liniei

    void setcolor(char c) {color = c;}
    char getcolor() {return color;}
    void setbkcolor(char c) {color = color | (c<<4);}
    char getbkcolor() {return (color>>4) & 127;}

    friend wintype &operator<<(wintype &o, char *s);
    friend wintype &operator>>(wintype &o, char *s);
};

// Construieste o fereastra.
wintype::wintype(int lx, int uy, // sus stinga
        int rx, int ly, // jos dreapta
```



```
        int b, // diferit de zero pentru chenar
        char *mesaj // mesaj pentru titlu
    )
{
    if(lx<0) lx = 0;
    if(rx>79) rx = 79;
    if(uy<0) uy = 0;
    if(ly>24) ly = 24;
    stgx = lx; susy = uy;
    drptx = rx, josy = ly;
    chenar = b;
    titlu = mesaj;
    activ = 0;
    cursx = cursy = 0;
    buf = new char[2*(drptx-stgx+1)*(josy-susy+1)];
    if(!buf) {
          cout << "Eroare de alocare.\n";
          exit(1);
    }
    color = alb;
}

// Afiseaza o fereastra
void wintype::winput()
{
    // activeaza fereastra
    if(!activ) { // nu este folosita in acel moment
          salv_ecran(); // salveaza ecranul curent
          activ = 1;
    }
    else return; // deja pe ecran

    if(chenar) trasat_chenar();
    afisat_titlu();

    // pozitioneaza cursorul in coltul din stanga sus
    goto_xy(stgx + cursx + 1, susy + cursy + 1);
}

// Indeparteaza fereastra si restaureaza continutul initial
// al ecranului.
void wintype::winremove()
{
```

```
    if(!activ) return; // nu poate sa indeparteze o
                       // fereastra inactiva
    restaur_ecran(); // restaureaza ecranul initial
    activ = 0; // restaurare video
}

// Traseaza un chenar in jurul ferestrei.
void wintype::trasat_chenar()
{
    register int i;
    char far *v, far *t;

    v = vid_mem;
    t = v;
    for(i=stgx+1; i<drptx; i++) {
          v += (susy*160) + i*2;
          *v++ = 196; *v = color;
          v = t;
          v += (josy*160) + i*2;
          *v++ = 196;
          *v = color;
          v = t;
    }
    for(i=susy+1; i<josy; i++) {
          v += (i*160) + stgx*2;
          *v++ = 179;
          *v = color;
          v = t;
          v += (i*160) + drptx*2;
          *v++ = 179;
          *v = color;
          v = t;
    }

    // traseaza colturile
    scrie_car(stgx, susy, 218, color);
    scrie_car(stgx, josy, 192, color);
    scrie_car(drptx, susy, 191, color);
    scrie_car(drptx, josy, 217, color);
}

// Afiseaza titlul ferestrei.
void wintype::afisat_titlu()
```



```
{
    register int x, lung;

    x = stgx;

    /* Calculeaza pozitia de pornire corecta pentru a centra
    titlul - daca este negativ, mesajul nu se va incadra.
    */
    lung = strlen(titlu);
    lung = (drptx - x - lung) / 2;
    if(lung<0) return; // nu il afiseaza
    x = x + lung + 1;
    scrie_sir(x, susy, titlu, color);
}

// Salveaza ecranul, el putand fi restaurat dupa
// indepartarea ferestrei.
void wintype::salv_ecran()
{
    register int i, j;
    char *buf_ptr;
    char far *v, far *t;

    buf_ptr = buf;
    v = vid_mem;
    for(i=susy; i<josy+1; i++) {
          for(j=stgx; j<drptx+1; j++) {
                t = (v + (i*160) + j*2);
                *buf_ptr++ = *t++;
                *buf_ptr++ = *t;
                *(t-1) = ' '; // sterge fereastra
    }
}

// Restaureaza o zona a ecranului.
void wintype::restaur_ecran()
{
    register int i, j;
    char far *v, far *t;
    char *buf_ptr;

    buf_ptr = buf;
    v = vid_mem;
```

```
    t = v;
    for(i=susy; i<josy+1; i++)
          for(j=stgx; j<drptx+1; j++) {
                v = t;
                v += (i*160) + j*2;
                *v++ = *buf_ptr++; // scrie caracterul
                *v = *buf_ptr++; // scrie atributul
    }
}

/* Scrie un sir in pozitia curenta a cursorului
   in fereastra specificata.
   Returneaza 0 daca fereastra nu este activa;
   altfel, returneaza 1.
*/
int wintype::winputs(char *s)
{
    register int x, y;
    char far *v;

    // se asigura ca fereastra este activa
    if(!activ) return 0;

    x = cursx + stgx + 1;
    y = cursy + susy + 1;

    v = vid_mem;
    v += (y*160) + x*2; // calculeaza adresa de pornire
    for( ; *s; s++) {
          if(y >= josy) {
                return 1;
          }

          if(x >= drptx) {
                return 1;
          }
          if(*s=='\n') {
                y++;
                x = stgx + 1;
                v = vid_mem;
                v += (y*160) + x*2; // calculeaza adresa
                cursy++; // incrementeaza Y
                cursx = 0; // aduce pe x la 0;
```



```
          }
          else {
                cursx++;
                x++;
                *v++ = *s; // scrie caracterul
                *v++ = color; // culoare
          }
          winxy(cursx, cursy);
    }
    return 1;
}

/* Plaseaza cursorul intr-o fereastra intr-o pozitie
   specificata. Returneaza 0 daca este in afara limitelor;
   altfel, diferit de zero.
*/
int wintype::winxy(int x, int y)
{
    if(x<0 || x+stgx >= right-1)
          return 0;
    if(y<0 || y+susy >= josy-1)
          return 0;
    cursx = x;
    cursy = y;
    goto_xy(left+x+1, susy+y+1);
    return 1;
}

// Citeste un sir dintr-o fereastra.
void wintype::wingets(char *s)
{
    char ch, *temp;

    temp = s;
    for(;;) {
     ch = wingetche();
     switch(ch) {
        case '\r': // este apasata tasta ENTER
           *s = '\0';
           return;
        case '\b': // backspace
           if(s > temp) {
                s--;
```

```
                cursx--;
                if(cursx < 0) cursx=0;
                winxy(cursx, cursy);
                scrie_car(stgx + cursx + 1, susy + cursy +
                        1, ' ', color);
            }
            break;
        default: *s = ch;
            s++;
        }
    }
}
/* Introduce o tasta apasata in interiorul unei ferestre.
   Returneaza codul complet pe 16 biti al tastei.
*/
wintype::wingetche()
{
    union inkey {
        char ch[2];
        int i;
    } c;
    union REGS r;
    if(!activ) return 0; // fereastra nu este activa

    winxy(cursx, cursy);

    r.h.ah = 0; // citeste o tasta
    c.i = int86(0x16, &r, &r);

    if(c.ch[0]); {
        switch(c.ch[0]) {
            case '\r': // este apasata tasta ENTER
                break;
            case '\b': // back space
                break;
            default:
                if(cursx + stgx < drptx - 1) {
                    scrie_car(stgx + cursx + 1,
                    susy + cursy + 1, c.ch[0], color);
                    cursx++;
                }
        }
        if(cursy < 0) cursy = 0;
```



```
            if(cursy + susy > josy - 2)
                cursy--;
            winxy(cursx, cursy);
    }
    return c.i;
}

// Sterge continutul ferstrei.
void wintype::wincls()
{
    register int i, j;
    char far *v, far *t;

    v = vid_mem;
    t = v;
    for(i=susy+1; i<josy; i++)
        for(j=stgx+1; j<drptx; j++ {
            v = t;
            v += (i*160) + j*2;
            *v++ = ' '; // scrie un spatiu
            *v = color; // in culoarea fundalului
        }
    cursx = 0;
    cursy = 0;
}

// Sterge pana la sfarsitul liniei.
void wintype::wincleol()
{
    register int i, x, y;

    x = cursx;
    y = cursy;
    winxy(cursx, cursy);

    for(i=cursx; i<drptx-1; i++)
        winputs(" ");
    winxy(x, y);
}
// Iesire intr-o fereastra.
wintype &operator<<(wintype &o, char *s)
{
    o.winputs(s);
```

```
    return o;
}

// Intrare de la o fereastra.
wintype &operator>>(wintype &o, char *s)
{
    o.wingets(s);
    return o;
}

main()
{
    char s[80];

    set_v_ptr(); // da pointerul pentru memoria video

    wintype w1(1, 10, 20, 20, 1, "Fereastra Mea #1");
    wintype w2(40, 1, 60, 20, 1, "Fereastra mea #2");
    wintype w3(40, 5, 60, 20, 1, "Fereastra mea #3");

    w1.winput();
    w2.winput();
    w1.setcolor(rosu);
    w2.setcolor(verde);
    w1 >> s;
    w1.winxy(0, 0);
    w1.winputs("Va salut\n");
    w1.winputs("Ferestrele sunt distractive");
    w1 << "\n";
    w1 >> s;
    w1 << "Acesta \neste " << "un test" << "\n";
    w2 << "acesta este un test";
    w2.winxy(3, 4);
    w2 << "la locatia 3, 4";
    w1 << "Acesta este un alt test pe care sa il vedeti\n";
    w1 >> s;
    w3.winput(); // se suprapune peste alta fereastra
    w3 >> s;
    w3.winremove();
    w1.winxy(0, 0);
    w1.setcolor (rosu);
    w1.setbkcolor (cian);
    w1.winputs("PROMPT: ");
    w1.setcolor (negru);
    w1 >>s;
    w2.winxy(0, 4);
    w2.setcolor (galben);
    w2.setbkcolor (verde);
    w2.winputs(s);
    w2 >> s;
    w1.wincls();
    w1.winxy(5, 0);
    w1.wincleol();
    w2 >> s;
    w1.winremove();
    w2.winremove();
    w1.winput ();
    w1 >> s;
    return 0;
}

void set_v_ptr()
{
    int vmod;

    vmod = mod_video();
    if((vmod!=2) && (vmod!=3) && (vmod!=7)) {
          cout << "Video trebuie sa fie in modul text cu 80
                  de coloane. ";
          exit(1);
    }
    // stabileste adresa corecta a memoriei RAM video
    if(vmod==7) vid_mem = (char far *) 0xB0000000;
    else vid_mem = (char far *) 0xB8000000;
}

// Returneaza modul video curent.
mod_video()
{
    union REGS r;

    r.h.ah = 15; // da modul video
    return int86(0x10, &r, &r) & 255;
}

// Scrie caracter cu atribut specificat.
```

```
void scrie_car(int x, int y, char ch, int atrib)
{
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2;
    *v++ = ch; // scrie caracterul
    *v = atrib; // scrie atributul
}

// Trimite cursorul in pozitia X,Y specificata.
void goto_xy(int x, int y)
{
    union REGS r;

    r.h.ah = 2; // functia de adresare catre cursor
    r.h.dl = x; // coordonata coloanei
    r.h.dh = y; // coordonata randului
    r.h.bh = 0; // pagina video
    int86(0x10, &r, &r);
}

// Afiseaza un sir cu atribut specificat.
void scrie_sir(int x, int y, char *p, int atrib)
{
    register int i;
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2; // calculeaza adresa
    for(i=y; *p; i++) {
          *v++ = *p++; // scrie caracterul
          *v++ = atrib; // scrie atributul
    }
}
```

Acest program produce ieşirea prezentată în **Figura 24-1**.

Lucruri de încercat

Chiar dacă sistemul de ferestre este complet funcţional, ar fi unele modificări pe care aţi putea să le încercaţi. Pentru moment, când o fereastră este închisă, conţinutul ei se pierde. Aţi putea să-i salvaţi conţinutul, astfel încât să poată fi

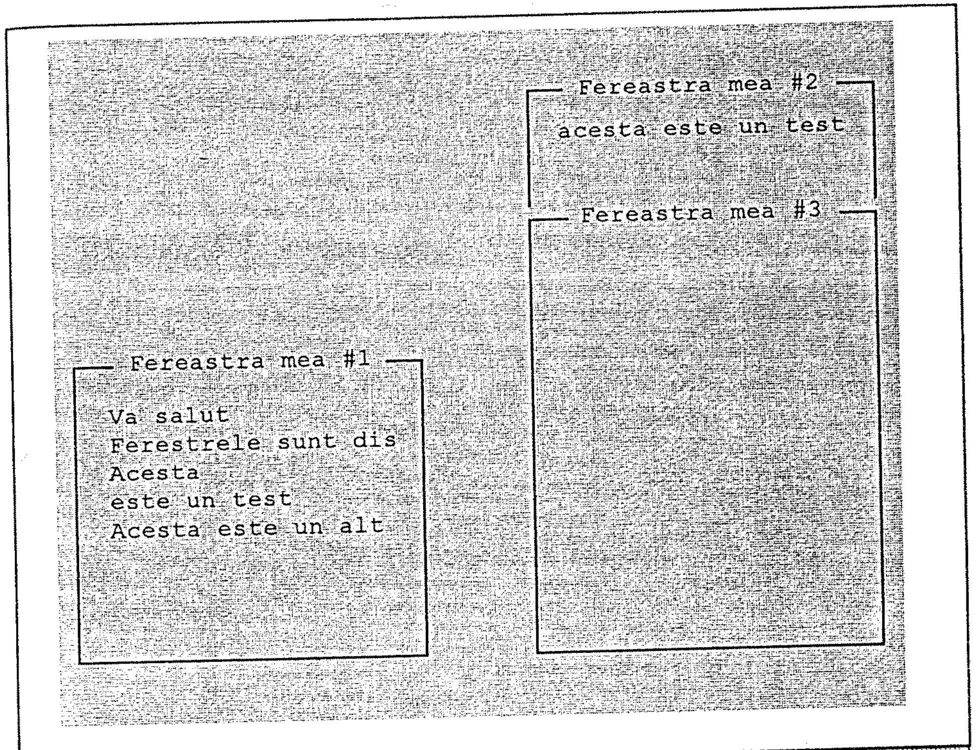


Figura 24-1. *Rezultatul programului exemplu pentru ferestre*

restaurat atunci când se reafişează fereastra. Această modificare poate fi de mare folos în anumite situaţii. Puteţi să supraîncărcaţi operatorii **<<** şi **>>**, astfel încât să poată lucra şi cu alte tipuri de date, nu numai cu şiruri. Puteţi supraîncărca operatorul = relativ la obiecte de tipul **wintype**. Dacă o faceţi, asiguraţi-vă că, pentru a păstra conţinutul unei ferestre, fiecare obiect foloseşte propria sa memorie. În sfârşit, dacă doriţi, puteţi utiliza un obiect de tip fereastră pentru a iniţializa un altul, va trebui însă să creaţi un constructor de copie care să determine ca fiecărui obiect să i se aloce o memorie proprie. Dacă nu o faceţi, atunci când se distrug obiectele va fi eliberată de două ori aceeaşi zonă de memorie.

Capitolul 25

O clasă generică de liste înlănţuite

Ultimul capitol al acestei cărţi examinează problemele care apar când creaţi o clasă generică de liste dublu înlănţuite. Clasele generice sunt unele dintre cele mai importante caracteristici din C++ - în special în mediile de programare profesionale. Chiar dacă o listă dublu înlănţuită este doar una dintre metodele multiple folosite pentru stocarea informaţiilor, problemele şi soluţiile asociate creării unei clase generice de acest tip pot fi generalizate pentru oricare metodă de memorare.

*REŢINEŢI: O clasă generică este una formată, folosind cuvântul cheie **template**. Tipul de date asupra căruia operează este specificat ca parametru la iniţializarea fiecărui obiect al clasei.*

Crearea unei clase generice de liste înlănţuite necesită folosirea unor caracteristici de C++ dintre cele mai avansate şi mai abstracte. Din acest motiv, capitolul va începe cu introducerea unei clase de liste duble înlănţuite care nu este generică. Ea creează o listă dublu înlănţuită pentru un anume tip de date, indicat explicit în program. Această versiune particulară a clasei de liste înlănţuite este folosită pentru a dezvolta, a prezenta şi a explica mecanismul de bază al listelor înlănţuite. Apoi, ea va fi transformată într-una generică, ce poate lucra cu orice tip de date.

O clasă simplă de liste dublu înlănţuite

După cum ştiţi, probabil, listele dublu înlănţuite sunt structuri de date dinamice care pot să îşi mărească sau să îşi micşoreze lungimea în timpul execuţiei programului. De fapt, principalul avantaj al unei structuri de date dinamice este acela că mărimea sa nu trebuie să fie fixată în timpul compilării, ci este liberă să se dezvolte sau să se comprime după necesităţi în timpul rulării. Fiecare obiect din listă conţine o legătură către obiectul precedent şi una către următorul. Obiectele sunt introduse şi distruse din listă prin modificarea corespunzătoare a legăturilor. Deoarece listele dublu înlănţuite sunt structuri de date dinamice, de obicei fiecărui obiect memoria îi este alocată dinamic. Acesta este cazul şi cu clasele de liste dublu înlănţuite dezvoltate în acest capitol.

Fiecare element memorat într-o listă dublu înlănţuită conţine trei părţi: un pointer spre următorul element din listă, unul spre elementul precedent şi informaţiile conţinute în listă. **Figura 25-1** prezintă o astfel de listă. Listele pot memora orice tip de date, inclusiv caractere, întregi, structuri, clase, uniuni ş.a.m.d. Clasa de liste dublu înlănţuite dezvoltată în acest paragraf memorează doar caractere (pentru uşurinţa ilustrării), dar ar fi putut fi folosit orice alt tip de date.

Lista dublu înlănţuită este introdusă folosind o simplă ierarhie de clase. O clasă numită **dbinlob** defineşte natura obiectelor care vor fi memorate în listă. Această

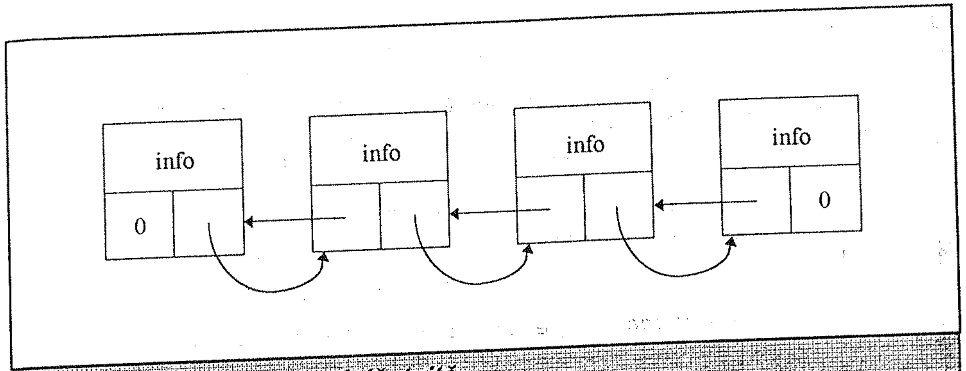


Figura 25-1 O listă dublu înlănţuită

clasă este apoi moştenită de alta, numită **dllist**, care introduce efectiv mecanismul listei dublu înlănţuite.

Clasa prezentată aici, **dbinlob**, defineşte natura fiecărui element din listă:

```
// Aceasta clasa defineste fiecare element din lista.
class dbinlob {
public:
    char info; // informatii
    dbinlob *urmator; // pointer spre obiectul urmator
    dbinlob *anterior; // pointer spre obiectul anterior
    dbinlob() {
          info = 0;
          urmator = NULL;
          anterior = NULL;
    }
    dbinlob(char c) {
          info = c;
          urmator = NULL;
          anterior = NULL;
    }
    dbinlob *daurmator() {return urmator;}
    dbinlob *daanterior() {return anterior;}
    void dainfo(char &c) { c = info;}
    void schimba(char c) { info = c; } // modifica un element

    // Supraincarca << pentru obiecte de tip dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob o)
    {
          stream << o.info << "\n";
          return stream;
```

```
    }

    // Supraincarca << pentru pointeri spre obiecte
    // de tip dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob *o)
    {
        stream << o->info << "\n";
        return stream;
    }

    // Supraincarca >> pentru referinte dbinlob.
    friend istream &operator>>(istream &stream, dbinlob &o)
    {
        cout << "Introduceti informatile: ";
        stream >> o.info;
        return stream;
    }
};
```

După cum puteţi vedea, **dbinlob** are trei membri de tip date. Membrul **info** păstrează informaţia memorată de către listă. Amintiţi-vă deocamdată tipul datelor este menţionat explicit în program drept **char**. De aceea, lista înlănţuită va fi capabilă să păstreze doar caractere. Pointerul **urmator** va indica spre următorul element al listei, iar **anterior** către elementul precedent. Reţineţi că membrii de tip date din **dbinlob** sunt publici. Ei sunt declaraţi astfel doar pentru prezentare şi pentru a permite ilustrarea mai uşoară a tuturor aspectelor listelor înlănţuite. În aplicaţiile dvs. puteţi să îi stabiliţi particulari sau protejaţi.

În **dbinlob** sunt definite, de asemenea, un număr de operaţii care pot fi efectuate asupra obiectelor de tip **dbinlob**. Anume, informaţiile asociate unui obiect pot fi extrase sau modificate şi pot fi obţinuţi pointerii spre elementul precedent şi următor. De asemenea, obiectele de tip **dbinlob** pot fi introduse şi obţinute la ieşire folosind operatorii suprapuşi << şi >>. Reţineţi că operaţiile definite în cadrul clasei **dbinlob** sunt independente de mecanismul de păstrare a listelor. **dbinlob** defineşte doar natura datelor care trebuie să fie memorate în listă.

Când este construit fiecare obiect, câmpurile **anterior** şi **următor** sunt iniţializate cu **NULL**. Pointerii rămân astfel până când obiectul este introdus într-o listă. Dacă este prezentă o valoare de iniţializare, ea este copiată în **info**, altfel **acesta** este iniţializat cu zero.

Funcţia **daurmator()** returnează un pointer spre elementul următor din listă. Dacă s-a ajuns la sfârşitul listei, acesta va fi **NULL**. Funcţia **daanterior()** întoarce un pointer spre elementul anterior din listă, dacă acesta există; altfel ea returnează **NULL**. Funcţiile nu sunt, practic, necesare deoarece **urmator** şi **anterior** sunt publice; veţi avea însă nevoie de ele când în aplicaţiile dvs. pointerii vor fi particulari.

Observaţi că operatorul << este supraîncărcat atât pentru obiecte de tip **dbinlob** cât şi pentru pointeri spre obiecte de tip **dbinlob**. Deoarece este uzual ca, atunci când folosiţi o listă înlănţuită, să aveţi acces la membrii listei folosind pointeri, este necesar să supraîncărcaţi << astfel încât să funcţioneze şi atunci când se transmite un pointer către obiect. Însă, deoarece nu există nici un motiv să interziceţi afişarea directă a unui obiect, a fost introdusă şi cea de-a doua formă, care operează direct asupra unui obiect.

Deşi **dbinlob** defineşte natura unei liste de obiecte înlănţuite, nu ea este cea care le creează. Mecanismul de înlănţuire a listelor este furnizat de clasa **dllist**, prezentată mai jos. După cum puteţi vedea, ea moşteneşte **dbinlob** şi operează asupra obiectelor de acest tip.

```
// Aceasta clasa introduce, de fapt, lista dublu inlantuita.
class dllist : public dbinlob {
    dbinlob *incep, *sfirs;
public:
    dllist() { incep = sfirs = NULL; }
    void memo(char c);
    void indep(dbinlob *ob); // scoate elementul
    void inceplist(); // afiseaza lista de la inceput spre
                      // sfarsit
    void sfarslist(); // afiseaza lista de la sfarsit spre
                      // inceput

    dbinlob *gaseste(char c); // returneaza pointer spre
                              // elementul cautat

    dbinlob *daincep() { return incep; }
    dbinlob *dasfirs() { return sfirs; }
};
```

Clasa **dllist** întreţine doi pointeri: unul către începutul listei, iar celălalt către sfârşitul ei. După cum puteţi vedea, aceştia sunt pointeri către obiecte de tip **dbinlob**. Aceşti pointeri sunt iniţializaţi cu NULL la crearea listei. Clasa **dllist** admite mai multe operaţii cu listele dublu înlănţuite, incluzând:

- Introducerea unui element în listă
- Scoaterea unui element din listă
- Parcurgerea listei de la început sau de la sfârşit
- Căutarea unui anumit element
- Obţinerea de pointeri către începutul sau către sfârşitul listei

În continuare este examinată fiecare procedură.

Funcţia memo()

Listei i se adaugă informaţii folosind funcţia **memo()**. Ea este introdusă după cum se vede aici:

```
// Adauga urmatoarea intrare.
void dllist::memo(char c)
{
    dbinlob *p;
    p = new dbinlob;
    if(!p) {
        cout << "Eroare de alocare.\n";
        exit(1);
    }

    p->info = c;

    if(incep==NULL) { // primul element din lista
        sfars = incep = p;
    }
    else { // pune la sfarsit
        p->anterior = sfars;
        sfars->urmator = p;
        sfars = p;
    }
}
```

Înainte ca un element să poată fi introdus în listă, trebuie creat de un obiect de tip **dbinlob** care să îl memoreze. Deoarece listele înlănţuite sunt structuri de date alocate dinamic, este normal ca **memo()** să obţină un obiect dinamic, folosind **new**. După ce s-a alocat memorie pentru un obiect de tip **dbinlob**, **memo()** atribuie informaţia transmisă prin **c** membrului **info** al noului obiect şi apoi adaugă obiectul la sfârşitul listei. Reţineţi că pointerii **incep** şi **sfars** sunt actualizaţi în concordanţă cu situaţia. În acest fel, **incep** şi **sfars** vor indica mereu către începutul şi, respectiv, sfârşitul listei.

Deoarece obiectele sunt adăugate întotdeauna la sfârşitul listei, lista nu este sortată. Puteţi modifica **memo()**, dacă doriţi, astfel încât să întreţină o listă sortată.

Funcţia indep()

Funcţia **indep()** scoate un obiect din listă. Ea este prezentată aici:


```
/* Indeparteaza un element din lista si reactualizeaza
   pointerii incep si sfars
*/
void dllist::indep(dbinlob *ob)
{
    if(ob->anterior) { // nu este vorba de primul element
        ob->anterior->urmator = ob->urmator;
        if(ob->urmator) // nu este vorba de ultimul element
            ob->urmator->anterior = ob->anterior;
        else // indeparteaza ultimul element
            sfars = ob->anterior; // reactualizeaza
                                  // pointerul sfirs
    }
    else { // indeparteaza primul element
        if(ob->urmator) { // lista nu este goala
            ob->urmator->anterior = NULL;
            incep = ob->urmator;
        }
        else // lista este goala acum
            incep = sfars = NULL;
    }
}
```

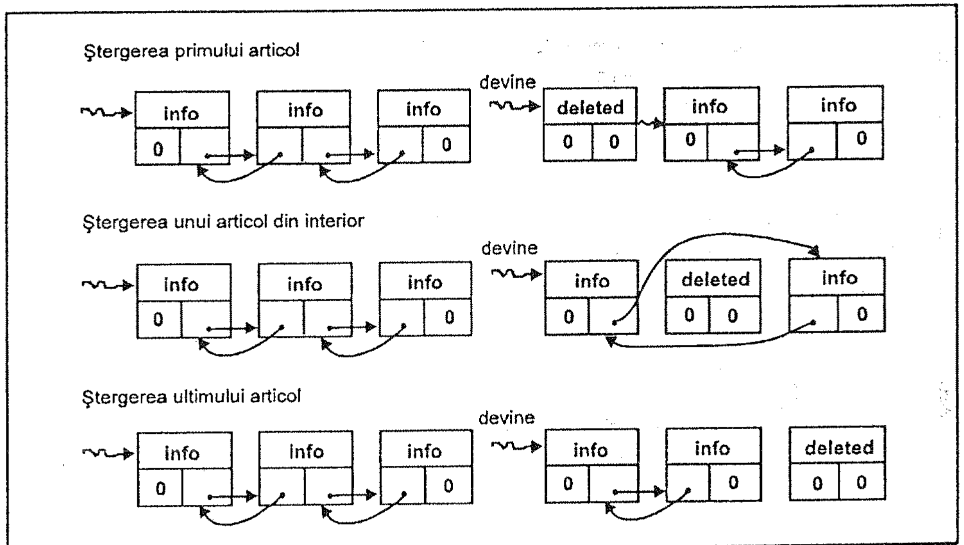


Figura 25-2. *Eliminarea unui obiect dintr-o listă dublu înlănţuită*

Funcţia **indep()** scoate din listă obiectul spre care indică parametrul său **ob**. (**ob** trebuie să fie un pointer valid spre un obiect de tip **dbinlob**.) Un obiect care trebuie scos poate să se găsească în trei locuri (vezi **Figura 25-2**). El poate fi primul element, ultimul sau undeva între acestea. Funcţia **indep()** tratează toate trei cazurile.

Reţineţi că funcţia scoate un obiect din listă, dar acesta nu este distrus. El este pur şi simplu „desprins din înlănţuire". (Desigur, puteţi să îl distrugeţi dacă doriţi, folosind **delete**.)

Ca şi **memo()**, operaţia efectuată de **indep()** nu depinde de tipul de date care sunt memorate efectiv în listă.

Afişarea listei

Funcţiile **inceplist()** şi **sfarslist()** afişează conţinutul listei de la început şi, respectiv, de la sfârşit. Aceste funcţii sunt incluse pentru a vă ilustra cum lucrează clasa **dllist**. Ele oferă un ajutor pentru depanarea programelor.

```
// Parcurge lista de la inceput spre sfarsit.
void dllist::inceplist()
{
   dbinlob *temp;

   temp = incep;
   do {
         cout << temp->info << " ";
         temp = temp->daurmator();
   }     while(temp);
   cout << "\n";
}

// Parcurge lista de la sfarsit catre inceput.
void dllist::sfirsit()
{
   dbinlob *temp;

   temp = sfirs;
   do {
         cout << temp->info << " ";
         temp = temp->daanterior();
   }     while(temp);
   cout << "\n";
}
```


Găsirea unui obiect din listă

Funcţia **gaseste()**, prezentată aici, returnează un pointer spre obiectul din listă care conţine informaţiile ce coincid cu cele specificate ca parametru. Ea va returna **NULL** dacă nu se găseşte nici un obiect corespunzător.

```
// Gaseste un obiect pe baza informatiilor sale.
dbinlob *dllist::gaseste(char c)
{
   dbinlob *temp;

   temp = incep;

   while(temp) {
         if(c==temp->info) return temp; // gasit
         temp = temp->daurmator();
   }
   return NULL; // nu este in lista
}
```

Un exemplu de program de listă dublu înlănţuită

Iată clasele complete **dbinlob** şi **dllist**, împreună cu o funcţie **main()** care ilustrează utilizarea lor:

```
// O clasa negenerica de liste dublu inlantuite.

#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class dbinlob {
public:
   char info; // informatii
   dbinlob *urmator; // pointer spre obiectul urmator
   dbinlob *anterior; // pointer spre obiectul anterior
   dbinlob() {
         info = 0;
         urmator = NULL;
         anterior = NULL;
   };
   dbinlob(char c) {
```

```
        info = c;
        urmator = NULL;
        anterior = NULL;
    }
    dbinlob *daurmator() {return urmator;}
    dbinlob *daanterior() {return anterior;}
    void dainfo(char &c) { c = info;}
    void schimba(char c) { info = c; } // modifica un element

    // Supraincarca << pentru obiecte de tip dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob o)
    {
        stream << o.info << "\n";
        return stream;
    }

    // Supraincarca << pentru pointeri spre obiecte de tip
    // dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob *o)
    {
        stream << o->info << "\n";
        return stream;
    }

    // Supraincarca >> pentru referinte dbinlob.
    friend istream &operator>>(istream &stream, dbinlob &o)
    {
        cout << "Introduceti informatiile: ";
        stream >> o.info;
        return stream;
    }
};

class dllist : public dbinlob {
    dbinlob *incep, *sfars;
public:
    dllist() { incep = sfars = NULL; }
    void memo(char c);
    void indep(dbinlob *ob); // scoate elementul
    void inceplist(); // afiseaza lista de la inceput spre
                      // sfarsit
    void sfarslist(); // afiseaza lista de la sfarsit spre
                      // inceput
```



```
    dbinlob *gaseste(char c); // returneaza pointer spre
                              // elementul cautat

    dbinlob *daincep() { return incep; }
    dbinlob *dasfirs() { return sfirs; }
};

// Adauga urmatoarea intrare.
void dllist::memo(char c)
{
    dbinlob *p;
    p = new dbinlob;
    if(!p) {
        cout << "Eroare de alocare.\n";
        exit(1);
    }

    p->info = c;

    if(incep==NULL) { // primul element din lista
        sfars = incep = p;
    }
    else { // pune la sfarsit
        p->anterior = p;
        sfars->urmator = p;
        sfars = p;
    }
}

/* Indeparteaza un element din lista si reactualizeaza
   pointerii incep si sfars
*/
void dllist::indep(dbinlob *ob)
{
    if(ob->anterior) { // nu este vorba de primul element
        ob->anterior->urmator = ob->urmator;
        if(ob->urmator) // nu este vorba de ultimul element
            ob->urmator->anterior = ob->anterior;
        else // indeparteaza ultimul element
            sfars = ob->anterior; // reactualizeaza
                                  // pointerul sfars
    }
```

```
    else { // indeparteaza primul element
          if(ob->urmator) { // lista nu este goala
                ob->urmator->anterior = NULL;
                incep = ob->urmator;
          }
          else // lista este goala acum
                incep = sfars = NULL;
    }
}

// Parcurge lista de la inceput spre sfarsit.
void dllist::inceplist()
{
    dbinlob *temp;

    temp = incep;
    do {
          cout << temp->info << " ";
          temp = temp->daurmator();
    }     while(temp);
    cout << "\n";
}

// Parcurge lista de la sfarsit catre inceput.
void dllist:: sfarslist()
{
    dbinlob *temp;

    temp = sfars;
    do {
          cout << temp->info << " ";
          temp = temp->daanterior();
    }     while(temp);
    cout << "\n";
}

// Gaseste un obiect pe baza informatiilor sale.
dbinlob *dllist::gaseste(char c)
{
    dbinlob *temp;

    temp = incep;
```



```
    while(temp) {
          if(c==temp->info) return temp; // gasit
          temp = temp->daurmator();
    }
    return NULL; // nu este in lista
}

main()
{
    dllist lista;
    char c;
    dbinlob *p;

    lista.memo('1');
    lista.memo('2');
    lista.memo('3');

    // foloseste functiile membre pentru a afisa lista
    cout << "Iata lista de la inceput, apoi de la sfarsit.\n";
    lista.inceplist();
    lista.sfarslist();

    cout << endl;

    // parcurge lista "manual"
    cout << "Parcurgere manuala a listei.\n";
    p = lista.daincep();
    while(p) {
          p->dainfo(c);
          cout << c << " ";
          p = p->daurmator(); // da urmatorul
    }

    cout << endl << endl;

    // cauta un element
    cout << "Cauta elementul 2.\n";
    p = lista.gaseste('2');
    if(p) {
          p->dainfo(c);
          cout << "Am gasit: " << c << endl;
    }
```

```
cout << endl;

// scoate un element
p->dainfo(c);
cout << "Scoate elementul " << c << ".\n";
list.indep(p);
cout << Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// adauga o alta intrare
cout << "Adauga un element.\n";
lista.memo('4');
cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// modifica informatiile
p = lista.gaseste('1');
if(!p) {
      cout << "Eroare, elementul nu a fost gasit.\n";
      return 1; // eroare
}

p->dainfo(c);
cout << "Modifica " << c << " in 5.\n";
p->modifica('5');
cout << "Iata lista de la inceput, apoi de la sfarsit.\n";
lista.inceplist();
lista.sfarslist();

cout << endl;

// ilustreaza << si >>
cin >> *p;
cout << p;

cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;
```



```
  // scoate primul element al listei
  cout << "Dupa indepartarea inceputului listei:\n";
  p = lista.daincep();
  lista.indep(p);
  lista.inceplist();

  cout << endl;

  // scoate ultimul element al listei
  cout << "Dupa indepartarea sfirsitului listei:\n";
  p = lista.dasfars();
  lista.indep(p);
  lista.inceplist();

  return 0;
}
```

Iată ieşirea produsă de acest exemplu. (Când programul a solicitat introducerea unei date, s-a scris **X**.)

```
Iata lista de la inceput, apoi de la sfarsit.
1 2 3
3 2 1

Parcurgere manuala a listei.
1 2 3

Cauta elementul 2.
Am gasit: 2

Scoate elementul 2.
Iata lista de la inceput.
1 3

Adauga un element.
Iata lista de la inceput.
1 3 4

Modifica 1 in 5.
Iata lista de la inceput, apoi de la sfarsit.
5 3 4
4 3 5
```

```
Introduceti informatiile: X
Iata lista de la inceput.
X 3 4

Dupa indepartarea inceputului listei:
3 4

Dupa indepartarea sfarsitului listei:
3
```

Crearea unei clase generice de liste dublu înlănţuite

Deşi clasa de liste creată în paragraful precedent este perfect validă, ea poate fi folosită doar pentru liste de caractere, deoarece acesta este tipul de date definite de către **dbinlob**. Dacă aţi vrea să memoraţi alte tipuri de date trebuie să modificaţi specificatorul de tip pentru **info** şi anumite funcţii pentru a le adapta la noul tip de date. Desigur, efectuarea acestor schimbări pentru fiecare tip nou de date este fastidios şi totodată sursă de erori. O soluţie mai bună este să creaţi o clasă generică de liste înlănţuite, folosind un şablon capabil să trateze automat orice tip de date. Exact acest lucru îl face acest paragraf.

Un avantaj de a crea o clasă generică de liste dublu înlănţuite este acela că detaşează mecanismul (adică algoritmul care întreţine lista înlănţuită) de datele memorate efectiv în listă. Astfel, mecanismul poate fi creat o singură dată şi refolosit de oricâte ori.

*NOTĂ: Noţiunile de bază despre crearea şi utilizarea unei clase generice sunt discutate în Capitolul 20. Dacă nu sunteţi familiarizat cu folosirea cuvântului cheie **template** sau cu clasele generice, în general, va trebui să citiţi acest capitol înainte de a încerca să înţelegeţi crearea unei clase generice de liste înlănţuite.*

Versiunea generică a clasei de liste înlănţuite

Primul pas pentru transformarea clasei **dbinlob** şi **dllist** în clase generice este să le declaraţi în şabloane. O dată făcut acest lucru, tipul de date asupra căruia vor opera este transmis ca parametru ori de câte ori este creat un obiect al acestei clase. Iată versiunile generice pentru **dbinlob** şi **dllist**:

```
template <class DataT> class dbinlob {
public:
```



```
DataT info; // informatie
dbinlob<DataT> *urmator; // pointer catre urmatorul obiect
dbinlob<DataT> *anterior; // pointer catre obiectul anterior
dbinlob() {
      info = 0;
      urmator = NULL;
      anterior = NULL
};
dbinlob(DataT c) {
      info = c;
      urmator = NULL
      anterior = NULL;
};

dbinlob<DataT> *daurmator() {return urmator;}
dbinlob<DataT> *daanterior() {return anterior;}
void dainfo(DataT &c) { c = info;}
void modif(DataT c) { info = c; } // modifica un element

// Supraincarca << pentru obiecte de tip dbinlob.
friend ostream &operator<<(ostream &stream,
                            dbinlob<DataT> o)
{
      stream << o.info << "\n";
      return stream;
}

// Supraincarca << pentru pointeri spre obiecte de tip
// dbinlob.
friend ostream &operator<<(ostream &stream,
                            dbinlob<DataT> *o)
{
      stream << o->info << "\n";
      return stream;
}

// Supraincarca >> pentru referinte dbinlob.
friend istream &operator>>(istream &stream,
                            dbinlob<DataT> &0)
{
      cout << "Introduceti informatiile: ";
      stream >> o.info;
      return stream;
```

```
    }
};

template <class DataT> class dllist : public dbinlob<DataT> {
    dbinlob<DataT> *incep, *sfirs;
public:
   dllist() { incep = sfars = NULL; }
   void memo(DataT c);
   void indep(dbinlob<DataT> *ob); // scoate elementul
   void inceplist(); // afiseaza lista de la inceput spre
                     // sfarsit
   void sfarslist(); // afiseaza lista de la sfarsit spre
                     // inceput
   dbinlob<DataT> *gaseste(DataT c); // returneaza pointer
                                     // spre elementul cautat
   dbinlob<DataT> *daincep() { return incep; }
   dbinlob<DataT> *dasfars() { return sfars; }
};
```

După cum puteţi vedea, tipul de date generic este numit **DataT**. El este folosit ca un specificator de tip pentru toate referinţele la datele memorate în **dbinlob**. Când este creat un obiect, acest tip este înlocuit cu tipul efectiv specificat.

De exemplu, pentru a crea o listă înlănţuită numită **listamea**, care poate memora valori de tip **unsigned long**, veţi folosi această declarare:

```
dllist<unsigned long> listamea;
```

Aceasta obţine un exemplar al versiunii pentru **dllist** care este capabil să memoreze întregi lungi fără semn. Fiţi atenţi în declaraţiile pentru **dbinlob** şi **dllist** la modul în care este tratat tipul generic **DataT** când **dbinlob** este moştenit de **dllist**. Anume, tipul de date folosit pentru exemplarul pentru **dllist** este pasat, de asemenea, către **dbinlob**. Astfel, în declararea anterioară tipul de date **unsigned long** este pasat clasei **dllist** care îl transmite mai departe clasei **dbinlob**. Aceasta înseamnă că, în situaţia respectivă, tipul de date memorat de un obiect de tip **dbinlob** va fi **unsigned long**.

Pentru a crea un alt tip de listă, modificaţi pur şi simplu specificarea tipului de date. De exemplu, următoarea instrucţiune creează o listă pentru memorarea pointerilor către variabile de tip caracter:

```
dllist<char *> CarPointList;
```


Clasa generică pentru liste dublu înlănţuite

În continuare este prezentată întreaga clasă generică pentru liste dublu înlănţuite şi un exemplu de funcţie **main()**. Reţineţi felul în care este folosit tipul de date generic în cadrul definirii funcţiilor. După cum puteţi vedea, în toate cazurile, datele din listă asupra cărora se operează au fost specificate cu ajutorul tipului generic **DataT**. Natura concretă a datelor nu este rezolvată până când nu se obţine un exemplar efectiv de listă în cadrul funcţiei **main()**.

```
// O clasa generica de liste dublu inlantuite.
#include <iostream.h>

#include <string.h>
#include <stdlib.h>
template <class DataT> class dbinlob {
public:
   DataT info; // informatii
   dbinlob<DataT> *urmator; // pointer spre obiectul urmator
   dbinlob<DataT> *anterior; // pointer spre obiectul anterior
   dbinlob() {
         info = 0;
         urmator = NULL;
         anterior = NULL;
   };
   dbinlob(DataT c) {
         info = c;
         urmator = NULL;
         anterior = NULL;
   }
   dbinlob<DataT> *daurmator() {return urmator;}
   dbinlob<DataT> *daanterior() {return anterior;}
   void dainfo(DataT &c) { c = info;}
   void schimba(DataT c) { info = c; } // modifica un element

   // Supraincarca << pentru obiecte de tip dbinlob.
   friend ostream &operator<<(ostream &stream,
                              dbinlob<DataT> o)
   {
         stream << o.info << "\n";
         return stream;
   }

   // Supraincarca << pentru pointeri spre obiecte de tip
```

```
    // dbinlob.
    friend ostream &operator<<(ostream &stream,
                               dbinlob<DataT> *o)
    {
          stream << o->info << "\n";
          return stream;
    }

    // Supraincarca >> pentru referinte dbinlob.
    friend istream &operator>>(istream &stream,
                               dbinlob<DataT> &o)
    {
          cout << "Introduceti informatiile: ";
          stream >> o.info;
          return stream;
    }
};

template <class DataT> class dllist : public dbinlob<DataT> {
    dbinlob<DataT> *incep, *sfirs;
public:
   dllist() { incep = sfars = NULL; }
   void memo(DataT c);
   void indep(dbinlob<DataT> *ob); // scoate elementul
   void inceplist(); // afiseaza lista de la inceput spre
                     // sfarsit
   void sfarslist(); // afiseaza lista de la sfarsit spre
                     // inceput

   dbinlob<DataT> *gaseste(DataT c); // returneaza pointer
                                     // spre elementul cautat

   dbinlob<DataT> *daincep() { return incep; }
   dbinlob<DataT> *dasfars() { return sfars; }
};

// Adauga urmatoarea intrare.
template <class DataT> void dllist<DataT>::memo(DataT c)
{
    dbinlob<DataT> *p;

    p = new dbinlob<DataT>;
    if(!p) {
```



```
          cout << "Eroare de alocare.\n";
          exit(1);
    }

    p->info = c;

    if(incep==NULL) { // primul element din lista
          sfars = incep = p;
    }
    else { // pune la sfarsit
          p->anterior = p;
          sfars->next = p;
          sfars = p;
    }
}

/* Indeparteaza un element din lista si reactualizeaza
   pointerii incep si sfars.
*/
template <class DataT>
               void dllist<DataT>::indep(dbinlob<DataT> *ob)
{
    if(ob->anterior) { // nu este vorba de primul element
          ob->anterior->urmator = ob->urmator;
          if(ob->urmator) // nu este vorba de ultimul element
                ob->urmator->anterior = ob->anterior;
          else // indeparteaza ultimul element
                sfars = ob->anterior; // reactualizeaza
                                      // pointerul sfirs
    }
    else { // indeparteaza primul element
          if(ob->urmator) { // lista nu este goala
                ob->urmator->anterior = NULL;
                incep = ob->urmator;
          }
          else // lista este goala acum
                incep = sfars = NULL;
    }
}

// Parcurge lista de la inceput spre sfarsit.
template <class DataT> void dllist<DataT>::inceplist()
{
```

```
    // adauga o alta intrare
    cout << "Adauga un element.\n";
    lista.memo('4');
    cout << "Iata lista de la inceput.\n";
    lista.inceplist();

    cout << endl;

    // modifica informatiile
    p = lista.gaseste('1');
    if(!p) {
          cout << "Eroare, elementul nu a fost gasit.\n";
          return -1; // eroare
    }

    p->dainfo(c);
    cout << "Modifica " << c << " in 5.\n";
    p->modifica('5');
    cout << "Iata lista de la inceput, apoi de la
              sfarsit.\n";
    lista.inceplist();
    lista.sfarslist();

    cout << endl;

    // ilustreaza << si >>
    cin >> *p;
    cout << p;

    cout << "Iata lista de la inceput.\n";
    lista.inceplist();
    cout << endl;

    // scoate primul element al listei
    cout << "Dupa indepartarea inceputului listei:\n";
    p = lista.daincep();
    lista.indep(p);
    lista.inceplist();

    cout << endl;

    // scoate ultimul element al listei
    cout << "Dupa indepartarea sfarsitului listei:\n";
```



```
    p = lista.dasfirs();
    lista.indep(p);
    lista.inceplist();

    return 0;
}
```

Programul precedent foloseşte lista dublu înlănţuită pentru a memora date de tip caracter şi este echivalent funcţional cu versiunea de program, prezentată mai devreme în acest capitol, care nu este generică. Dar, motivul pentru a face clasele de liste înlănţuite să fie generice a fost să le permitem să fie folosite cu orice tip de date. Pentru a vedea cât de uşor este acest lucru, încercaţi să înlocuiţi funcţia **main()** din programul anterior cu următoarea, care creează o listă dublu înlănţuită de valori de tip **double**.

```
main()
{
    dllist<double> lista; // creeaza o lista in dubla precizie
    double c;
    dbinlob<double> *p;

    lista.memo('1.1');
    lista.memo('2.2');
    lista.memo('3.3');

    // foloseste functiile membre pentru a afisa lista
    cout << "Iata lista de la inceput, apoi de la
              sfarsit.\n";
    list.inceplist();
    list.sfarslist();

    cout << endl;

    // parcurge lista "manual"
    cout << "Parcurgere manuala a listei.\n";
    p = lista.daincep();
    while(p) {
          p->dainfo(c);
          cout << c << " ";
          p = p->daurmator(); // da urmatorul
    }

    cout << endl << endl;
```

```
    dbinlob<DataT> *temp;

    temp = incep;
    do {
          cout << temp->info << " ";
          temp = temp->daurmator();
    }     while(temp);
    cout << "\n";
}

// Parcurge lista de la sfarsit catre inceput.
template <class DataT> void dllist<DataT>::sfarslist()
{
    dbinlob<DataT> *temp;

    temp = sfars;

    do {
          cout << temp->info << " ";
          temp = temp->daanterior();
    }     while(temp);
    cout << "\n";
}

// Gaseste un obiect pe baza informatiilor.
template <class DataT> dbinlob<DataT>
                          *dllist<DataT>::gaseste(DataT c)
{
    dbinlob<DataT> *temp;

    temp = incep;

    while(temp) {
          if(c==temp->info) return temp; // gasit
          temp = temp->daurmator();
    }
    return NULL; // nu este in lista
}

main()
{
    dllist<char> lista;
    char c;
```



```
    dbinlob<char> *p;

    lista.memo('1');
    lista.memo('2');
    lista.memo('3');

    // foloseste functiile membre pentru a afisa lista
    cout << "Iata lista de la inceput, apoi de la
              sfarsit.\n";
    list.inceplist();
    list.sfarslist();

    cout << endl;

    // parcurge lista "manual"
    cout << "Parcurgere manuala a listei.\n";
    p = lista.daincep();
    while(p) {
          p->dainfo(c);
          cout << c << " ";
          p = p->daurmator(); // da urmatorul
    }

    cout << endl << endl;

    // cauta un element
    cout << "Cauta elementul 2.\n";
    p = lista.gaseste('2');
    if(p) {
          p->dainfo(c);
          cout << "Am gasit: " << c << endl;
    }

    cout << endl;
    // scoate un element
    p->dainfo(c);
    cout << "scoate elementul " << c << ".\n";
    lista.indep(p);
    cout << Iata lista de la inceput.\n";
    lista.inceplist();

    cout << endl;
```

```
// cauta un element
cout << "Cauta elementul 2.2.\n";
p = lista.gaseste(2.2);
if(p) {
      p->dainfo(c);
      cout << "Am gasit: " << c << endl;
}

cout << endl;

// scoate un element
p->dainfo(c);
cout << "Indeparteaza elementul " << c << ".\n";
lista.indep(p);
cout << Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// adauga o alta intrare
cout << "Adauga un element.\n";
lista.memo(4.4);
cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

//. modifica informatiile
p = lista.gaseste(1.1);
if(!p) {
      cout << "Eroare, elementul nu a fost gasit.\n";
      return 1; // eroare
}

p->dainfo(c);
cout << "Modifica " << c << " in 5.5.\n";
p->modifica(5.5);
cout << "Iata lista de la inceput, apoi de la
         sfarsit.\n";
lista.inceplist();
lista.sfarslist();
```



```
cout << endl;

// ilustreaza << si >>
cin >> *p;
cout << p;

cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// Scoate primul element al listei
cout << "Dupa indepartarea inceputului listei:\n";
p = list.daincep();
lista.indep(p);
lista.inceplist();

cout << endl;

// scoate ultimul element al listei
cout << "Dupa indepartarea sfarsitului listei:\n";
p = lista.dasfars();
lista.indep(p);
lista.inceplist();

return 0;
}
```

După ce scrieţi această funcţie **main()** în program, ea va determina următoarea ieşire:

```
Iata lista de la inceput, apoi de la sfarsit.
1.1 2.2 3.3
3.3 2.2 1.1

Parcurgere manuala a listei.
1.1 2.2 3.3

Cauta elementul 2.2
Am gasit: 2.2

Scoate elementul 2.2.
Iata lista de la inceput.
1.1 3.3
```

```
Adauga un element.
Iata lista de la inceput.
1.1 3.3 4.4

Modifica 1.1 in 5.5.
Iata lista de la inceput, apoi de la sfarsit.
5.5 3.3 4.4
4.4 3.3 5.5

Introduceti informatiile: 99.99
Iata lista de la inceput.
99.99 3.3 4.4

Dupa indepartarea inceputului listei:
3.3 4.4

Dupa indepartarea sfarsitului listei:
3.3
```

Ar trebui să încercaţi acum singuri să creaţi liste pentru alte tipuri de date. Amintiţi-vă că pot fi memorate în listă chiar şi tipurile de date compuse, cum ar fi structurile care conţin o adresă poştală.

Alte implementări

Există multe feluri în care poate fi utilizată o clasă de liste înlănţuite. Puteţi să faceţi încercări şi pe cont propriu. Iată câteva idei cu care aţi putea să începeţi.

Listele din acest capitol adaugă, pur şi simplu, obiecte la sfârşitul listei. Acest lucru este acceptabil (ba chiar de dorit) pentru multe aplicaţii. Totuşi, aţi putea modifica **memo()** astfel încât să creeze o listă sortată sau crea o altă versiune care să adauge elemente la începutul listei. De fapt, puteţi să definiţi mai multe versiuni ale funcţiei **memo()** (fiecare cu propriul său nume, care să o explice) ce memorează elemente în diferite feluri. De exemplu, puteţi defini funcţii denumite **MemoSfars()**, **MemoIncep()** şi **MemoSort()**, care să adauge elemente la sfârşit, la început sau, respectiv, într-o anumită ordine.

O funcţie pe care poate doriţi să o adăugaţi este numită **dalungime()**. Faceţi-o să returneze numărul de elemente din listă.

Aşa cum s-a menţionat mai devreme, pointerii **urmator** şi **anterior** au fost declaraţi publici intenţionat, pentru a simplifica rutinele pentru listele înlănţuite şi pentru a ilustra complet clasele de liste înlănţuite. Puteţi însă stabili ca aceşti pointeri să fie particulari, protejându-i astfel împotriva utilizărilor greşite.

Un ultim lucru: chiar dacă listele înlănţuite prezentate în acest capitol memorează caractere şi numere în virgulă mobilă, amintiţi-vă că poate fi memorat orice tip de date.

Anexa A

Bibliotecile de clase standard propuse

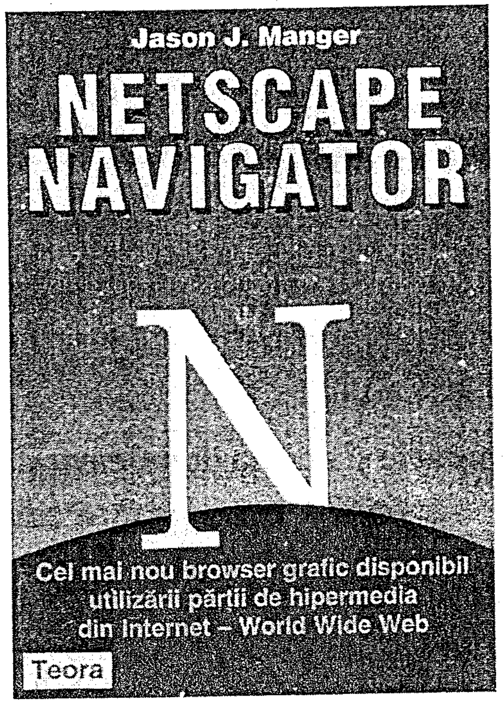
Comitetul de standardizare C++ este în procesul de definitivare a setului de biblioteci de clase standard. Acum, în momentul scrierii acestei cărţi, ele sunt încă în lucru şi nu sunt acceptate complet de nici un compilator de C++ disponibil. De aceea, nu este oportun să discutăm bibliotecile de funcţii în această ediţie a cărţii. (Excepţie face doar biblioteca de I/O, care este introdusă curent în toate compilatoarele şi care este discutată în amănunţime în **Partea a doua**.) Totuşi, deoarece compilatoarele viitoare le vor conţine, este important pentru dvs. să cunoaşteţi ce veţi avea la dispoziţie. Iată, deci, lista cu bibliotecile care sunt definite curent de către standardul propus ANSI C++:

- Sprijin pentru limbaj
- Diagnostice
- Utilizare generală
- Şiruri
- Localizări
- Recipiente
- Iteratori
- Algoritmi
- Numerice
- Intrări/Ieşiri (I/O)

Chiar dacă standardul ANSI C++ este încă în stadiu de dezvoltare, ar fi bine să verificaţi manualul compilatorului dvs. pentru a vedea care dintre aceste biblioteci de clase sunt admise.

REŢINEŢI: Bibliotecile de clase se adaugă bibliotecii de funcţii standard, care este inclusă în toate compilatoarele de C++.

De la editura McGraw-Hill s-a mai tradus:



Iată o carte despre cum se foloseşte eficient browserul Netscape Navigator! Prin Netscape puteţi vedea sistemul Internet printr-o interfaţă grafică atrăgătoare pentru utilizatorul final. Veţi găsi aici instrucţiuni şi detalii despre cum se instalează şi de configurează Netscape, cum se creează şi se publică prin HTML propriile pagini excelente de Web, cum se utilizează şi se percepe interfaţa Netscape, cum se caută informaţiile. Cartea mai prezintă limbajul HTML îmbunătăţit şi are o serie de anexe cu liste de URL-uri, index de aplicaţii ajutătoare, un glosar Web şi o secţiune de întrebări şi răspunsuri despre Netscape.

Teora - Cartea prin poștă

Peste 100.000 cititori (martie 1998) beneficiază deja de acest sistem. Lunar, alte câteva mii de noi cititori apelează la serviciile noastre.

Puteți primi la domiciliu cărțile dorite, cu plata ramburs la primirea coletului!

Tot ce aveți de făcut este să solicitați, printr-o simplă scrisoare, buletinul informativ al editurii noastre, care vă va fi expediat gratuit.

Precizați numele și adresa dumneavoastră. Veți avea astfel prilejul să fiți informat asupra titlurilor Teora disponibile și asupra celor în curs de apariție, pe care le veți putea comanda.

Nu ezitați! Contactați-ne acum pe adresa:

Editura Teora – Cartea prin poștă,
CP 79-30, București

sau telefonați la: **252.14.31**

Nu uitați! Teora vă pregătește pentru secolul 21